INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A programming language and operating system for parallel processing computing machines. (Volumes I and II)

Bergmann, Gregory John, Ph.D.
The Ohio State University, 1992
A PROGRAMMING LANGUAGE AND OPERATING SYSTEM FOR PARALLEL PROCESSING COMPUTING MACHINES

VOLUME I

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Gregory John Bergmann, B.S.A.R., M.S.

* * * * *

The Ohio State University

1992

Dissertation Committee: Approved by

J. M. Jagadeesh

H. Hemami

S. Ahalt

Advisor

Department of Electrical Engineering
DEDICATION

This work is dedicated in memory of

my brother, Douglas R. Bergmann
I express sincere appreciation to each member of our research group for the assistance they provided throughout this project. In particular, I extend thanks to Dr. Inching Chen, Dileep Krishna, and Mohammad Nikuie for their valuable contributions to the Pyramid Processing System project. And of course, I thank Dr. Dan Kramer, both for his nutritional advice and for making his vast if highly technical statistics library available to me. McDLT's and matrices will never be the same.

Thanks are due to The Ohio State University College of Pharmacy and to The Ohio State University Comprehensive Cancer Center for providing me with financial support during a portion of my graduate studies.

I want to express my appreciation to Professor Hooshang Hemami and to Professor Stanley Ahalt for the time and help they provided as members of the dissertation committee.

I particularly want to thank Professor Jogikal M. Jagadeesh, my advisor and friend, for the tremendous amount of support and guidance he provided me over the last several years.
The support of my family throughout my graduate school endeavors has been invaluable is and greatly appreciated. Finally, I thank my fiance, Stacey, for her patience and for understanding my frequent absences during the term of this project.
VITA

June 11, 1962 ................ Born - Marion, Ohio

1980 ......................... High School Diploma,
Marion Harding High School,
Marion, Ohio

1983-1987 ..................... Manager,
Recording Studio Facilities,
The Ohio State University
School of Music,
Columbus, Ohio

1984 ........................... B.S. Audio Recording,
The Ohio State University,
Columbus, Ohio

1985-1987 ..................... Graduate Research Associate,
The Ohio State University
College of Pharmacy,
Columbus, Ohio

1985-1986 ..................... Vice Chairman
The Audio Engineering Society
The Ohio State University Section

1986 ........................... M.S. Electrical Engineering,
The Ohio State University,
Columbus, Ohio

1986-1987 ..................... Chairman
The Audio Engineering Society
The Ohio State University Section
1987-1988  . . . . . . . . . . . . . . . . . . . Graduate Research Associate
                      Comprehensive Cancer Center
                      The Ohio State University,
                      Columbus, Ohio

1988-Present  . . . . . . . . . . . . . . . . . Technical Director
                      NexTech, Inc.
                      Columbus, Ohio

FIELDS OF STUDY

Field of Major : Electrical Engineering
Major Area : Digital Systems
Minor Area : Control Systems
Minor Area : Computer Information Science

PUBLICATIONS


# TABLE OF CONTENTS

DEDICATION ................................................................. ii

ACKNOWLEDGEMENTS .................................................. iii

VITA .................................................................................. v

LIST OF TABLES .................................................................. xi

**Chapter I Introduction** .................................................. 1
  1.1 Introduction ................................................................ 1
  1.2 Research Goals ....................................................... 4
  1.3 Dissertation Organization .......................................... 5

**Chapter II Literature Review** .......................................... 8
  2.1 Parallel Programming Languages ............................... 8
    2.1.1 Compiler Detected Parallelism .............................. 8
    2.1.1.1 Data Dependency Analysis ............................... 10
    2.1.1.2 Parallel Processing Granularity ....................... 20
    2.1.2 User Specified Parallelism .................................. 23
      2.1.2.1 Occam ....................................................... 23
      2.1.2.2 Concurrent Pascal ...................................... 26
      2.1.2.3 C Language Extensions for the Intel iPSC .......... 28
      2.1.2.4 Ada ......................................................... 31
      2.1.2.5 Modula 2 ................................................... 34
      2.1.2.6 LC3 .......................................................... 36
      2.1.2.7 Concurrent C .............................................. 38
      2.1.2.8 Linda ....................................................... 39
      2.1.2.9 CSP .......................................................... 41
      2.1.2.10 Parallel Pascal .......................................... 43
  2.2 Operating Systems ................................................... 44
  2.3 Summary of Literature Review .................................... 55
  2.4 Goals of Dissertation Project ...................................... 57
  2.5 BPL Implementation of the Dining Philosophers Problem .... 60
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - Edge Preserving Smoothing Timings</td>
<td>325</td>
</tr>
<tr>
<td>2 - Cumulative Spectral-Decay Calculation Timings</td>
<td>328</td>
</tr>
<tr>
<td>3 - Circuit Analysis Program Timing</td>
<td>330</td>
</tr>
<tr>
<td>4 - Image Compression Program Timing</td>
<td>334</td>
</tr>
</tbody>
</table>
Chapter I

Introduction

1.1 Introduction

Many current areas of computer research, including digital image processing, finite element modeling, analog system simulation, and statistical analysis, require the computational power of a supercomputer in order to achieve accurate calculated results in a reasonable amount of time. Unfortunately, the prohibitive cost of supercomputers prevents their presence in a typical research facility. Parallel processing is widely recognized as a cost effective method for achieving supercomputer performance. Furthermore, the speed limits of today's integrated circuit technology have nearly been reached. Therefore, parallel processing must be used if machines are to be built in the near future which are several orders of magnitude faster than the fastest computers of today.

A new generation of software tools must be used in order to utilize the power of parallel processing machines. The significance of advanced software in determining the success of computing hardware can be realized by examining the history of sequential computers. High level sequential programming languages, such as Fortran, Basic, Pascal, and C, simplified the task of programming
sequential machines by assisting programmers with tasks such as memory allocation, register usage, input and output operations, and subroutine linkage. Prior to the development of these high level languages, programmers were forced to perform these tasks "manually" using assembly language, a method which is difficult, time consuming, and error prone. Furthermore, programming in assembly language results in code which cannot be easily transported to different machines. High level languages permitted the programmer to develop programs without considering the details of the target machine's architecture. Powerful programs could be developed in a reasonable amount of time for execution on a wide variety of hardware.

Parallel processing programming languages must assist the programmer with a variety of tasks unique to parallel processing, including the creation of processes, synchronization of concurrent processes, communication between processes, and control of accesses to shared variables. The first attempts at programming parallel machines involved the use of existing sequential programming languages augmented with subroutines which invoked processes and allowed processes to communicate. The programmer was responsible for assigning processes to physical processors, determining routes for communication between processes, and synchronizing accesses to shared variables, all through the use of low level primitives. In many respects, this method of programming parallel processing machines is very similar to the programming of sequential machines using assembly language. The programmer is forced to consider details of the
target machine architecture, such as the number of processors and the interconnection scheme. Programs developed for a particular machine are not, in general, easily transported to different architectures. The programmer often must spend more time considering the control required for the architecture rather than developing the actual algorithm for the programming task.

Unfortunately, many parallel machines are still programmed in this manner.

Other methods exist for expressing parallel processing algorithms. One scheme involves the use of a compiler which is able to detect parallelism inherent in programs written in a standard sequential language. This approach offers the advantage that the programmer need not learn a new language. Additionally, existing programs need not be modified or rewritten in order to execute them on the new parallel machines. Unfortunately, compilers are seldom successful in automatically translating a sequentially expressed program into an efficient parallel algorithm. The programmer is frequently required to specify numerous data dependencies within the programs and make other modifications, thereby indirectly expressing a parallel algorithm. Programming a parallel machine in a sequential language has the further disadvantage that programmers are required to express algorithms in terms of a standard Von Neumann architecture, rather than develop new programming methodologies which reflect the underlying parallel architecture.

Parallel languages exist which allow the programmer to explicitly express a parallel algorithm. Some of these languages are modifications of existing languages, incorporating new control structures and data types which facilitate the
expression of parallelism. Others are new languages, designed specifically for parallel programming. These methods of programming are advantageous in that the programming can directly specify the concurrency within a parallel algorithm, thereby freeing him of the sequential model of programming and encouraging the development of new parallel programming methodologies. Modifications of existing languages are desirable since the programmer need not learn an entirely new language.

Some problems also exist with parallel programming languages. Many are designed for specific architectures, such as shared memory or communication network inter-connected machines. Therefore, programs written for one type of machine cannot easily be transported to another. Parallel languages are a relatively recent development. Several different schemes for expressing concurrency have been developed, each with supporters and critics. It is appropriate at this early point in the development of parallel languages to conduct further research into language structures for the expression of parallel algorithms for a broad class of machines.

1.2 Research Goals

The goal of this dissertation is to implement a parallel programming language, BPL, and an associated parallel operation system, BOS, on the Pyramid Processing System, described in [Chen, 1989]. This system is under development at The Ohio State University. The Pyramid is a shared memory machine which
will, in the future, also use an inter-connection communication network. BPL is an extension of the sequential language Pascal. It incorporates constructs which facilitate the use of the parallel machine resources of the Pyramid. It is desired, however, that the language be suitable for use with other architectures, such as those without shared memory or a communication network. The language constructs therefore are designed in a manner such that no specific requirements are imposed on the machine architecture.

Another goal of this research is for all system resource allocation to be performed automatically by the operating system. This will relieve the programmer of the task of assigning processes to physical processors. It will also facilitate the use of expanded or modified machine topologies without modification to application software.

The implementation of the language involves three distinct tasks. The first of these is the specification of the language, BPL. After the language is specified, the second task is to write a compiler to translate programs written in BPL into machine language which can be directly executed on the target machine. A recursive descent compiler is created for this task. Finally, an operating system, BOS, must be written in order to support the execution of programs written in BPL.

1.3 Dissertation Organization

Chapter II is a literature review of material relevant to this project. This
review is divided into two sections, corresponding to the two sections of the research. The first review section deals with programming languages for parallel processing systems. The second section discusses literature related to parallel processing operating systems.

Chapter III is a specification of the implemented language, BPL. In this chapter, all extensions to the Pascal language are described in detail. Brief examples are offered which demonstrate the use of the new language constructs.

Chapter IV is a description of the compiler. This chapter includes a detailed discussion of the different portions of the compiler. Pseudo code versions of the compiler code are presented.

Chapter V describes the BOS operating system used to support the execution of programs written in the BPL language. This chapter describes the various data structures used by the operating system. Pseudo code is also presented to illustrate the action of the various operating system routines.

Chapter VI details several sample programs written in the language. In order to demonstrate the automatic allocation of system resources, the execution time for these programs is measured for a variety of different system configurations.

Concluding remarks are found in Chapter VII. This chapter also discusses possible continued work on this project, including further compiler functions and program development tools.

Three appendices are included which contain the source code for the BPL
compiler, the BOS operating system, and the sample programs, respectively.
Chapter II

Literature Review

This chapter presents a literature review of publications and existing work related to programming systems for parallel processing. The review consists of discussions related to parallel processing programming languages and to parallel processing operating systems. At the conclusion of the review, the current state of parallel processing programming systems is discussed. Accordingly, appropriate objectives and goals for the dissertation project are presented. Finally, the contribution of the work is detailed.

2.1 Parallel Programming Languages

This section of the literature review shall discuss the language methods currently used to program parallel processing machines. The discussion concentrates on two different approaches to parallel program generation: compiler detected parallelism and programmer specified parallelism.

2.1.1 Compiler Detected Parallelism

Software systems which convert programs written in sequential languages
into equivalent parallel programs, as discussed in [Brent,1973], [Brode,1981], [Davis,1981], [Kruskal,1985], [Moiske,1985], [Muller,1976], [Padua,1986], [Cowell,1990], [Groww,1990], [Luecke,1991], and [Uresin,1990], posses many advantages. Much useful software, written in languages such as Fortran, is currently used in industry and research applications. By automatically transforming these programs into parallel code which can be executed on faster, more powerful computers, the cost of developing new software can be reduced. Furthermore, the use of existing sequential languages allows programmers to write software for new parallel processing machines without learning a new language or programming methodology.

Advanced compiler techniques are used to translate programs, written in a standard sequential language such as Fortran, into equivalent parallel programs which can be executed on a multi-processor machine. A variety of compilers of this type exist which produce parallel code representative of several different parallel models of execution. All such compilers, however, utilize the same basic program analysis techniques to determine which portions of a sequential program can be concurrently evaluated in a parallel program, producing results identical to those of the original sequential program.

This review first discusses the analysis techniques common to all parallelizing compilers. This is followed by a comparison of the methodologies used by the different classes of these compilers.
2.1.1.1 Data Dependency Analysis

[Brent, 1973], [Brode, 1981], [Muller, 1976], and [Padua, 1986] discuss the concepts of data dependence and the role of data dependencies in partitioning a sequential program onto multiple processors. Three types of data dependence exist in sequential programs. These are flow dependence, output dependence, and antidependence. Consider the following program statements:

\[
\begin{align*}
S1 & : X = Y \times Z \\
S2 & : W = X - Y
\end{align*}
\]

In S1, a value is calculated and assigned to the variable X. In S2, the value of X is used to calculate a new value, which is assigned to the variable W. For these statements, S1 must be evaluated before S2. Otherwise, an inaccurate value would be assigned to the variable W. This is an example of flow dependence. The following program statements illustrate output dependence:

\[
\begin{align*}
S1 & : X = Y \times Z \\
S2 & : W = X - Y \\
S3 & : X = 3 \times W
\end{align*}
\]

In this example, both S1 and S3 assign a new value to X. During the sequential execution of these statements, the final value of X is the value assigned by S3. Therefore, in order to obtain identical results, statement S3 must be evaluated after statement S1. This is an example of output dependence. Antidependence is illustrated below:
\[
S1 : W = X - Y \\
S2 : X = Y \times Z
\]

In this example, \(S2\) assigns a new value to a variable whose value is used in \(S1\). Therefore, in order to obtain the correct results, statement \(S1\) must be executed prior to \(S2\).

These types of data dependencies can be used to determine program statements which can be evaluated in parallel. Consider the following program statements:

\[
S1 : X = Y + Z \\
S2 : T = X + Y \\
S3 : X = 3 \\
S4 : A = B + C \\
S5 : C = Z + Y \\
S6 : A = C - B
\]

Due to a flow dependence, \(S1\) must be evaluated before \(S2\). Due to an antidependence, \(S2\) must be evaluated before \(S3\). Due to an output dependence, \(S1\) must be evaluated before \(S3\). Therefore, the first three statements must be executed sequentially. A similar analysis will reveal that statements \(S4\), \(S5\), and \(S6\) also must be executed sequentially. There are, however, no data dependencies between the first three statements and the final three statements. Therefore, a parallelizing compiler could generate code which would cause the first three statements to be executed on one processor and the final three statements on a different processor. This partitioning of the program will produce identical results to the sequential execution of the same program.
Special consideration must be given to array variables when determining data dependencies. A conservative approach would be to treat an entire array in the same manner as a simple scaler variable; no determination is made with regard to the values of array element indices. Therefore, a write to any element of an array would be treated as though the entire array were written. Any subsequent read of an array element would have to be executed sequentially due to the possible flow dependence.

A scheme which involves analysis of array indices is both desirable and feasible. The desirability is due to the fact that many programs spend the majority of their execution within loops, where identical operations are often performed on individual array elements. The feasibility arises from the fact that array references within loops often involve loop control variables; these variables behave in a very controlled manner, allowing the simplification of assumptions regarding successive evaluations of array index values. (It should be noted that these analysis techniques are best applied to DO and FOR loops; WHILE loops and loops with recursion are much more difficult to analyze [Padua,1986]).

The goal in analyzing loop statements is to determine if individual loop iterations can be executed concurrently. When determining data dependencies in loop constructs, dependencies between loop iterations must be examined. Consider the following loop construct:

```
DO 10 I = 1,1000
A(I) = B(I+1)
10 CONTINUE
```
Because the array element B(I+1) is never written to, no flow dependencies can occur. Because the array element A(I) is written to only once per iteration and the index variable is always unique, no antidependencies or output dependencies can occur. Therefore, there are no data dependencies between loop iterations, allowing, for example, each loop iteration to be executed concurrently on parallel processors.

The following loop statement, however, does contain data dependencies between loop iterations:

```
DO 20 I = 2,1000
   A(I) = A(I) + A(I-1)
20   CONTINUE
```

In this loop, each iteration reads the value of an array element which was assigned during the previous loop iteration. Therefore, iteration N cannot proceed until iteration N-1 has completed. This flow dependence prevents parallel execution of each iteration; the loop statement must be executed sequentially.

Very often, subroutine calls appear within loop statements. This can have detrimental effects on the success of generating parallel code. The reason for this is that subroutines can alter global variables. If a subroutine is called once within each loop iteration and the subroutine alters the value of a global variable, an output dependence exists between each loop iteration. Flow dependencies and antidependencies can also arise if the subroutine reads from and writes to global variables. Many compilers will not attempt to generate parallel code for loops
which contain subroutine calls. Others, such as PTOOL ([Allan,1986]), examine the subroutine call for global variable modification; if no global variable is modified, the subroutine code can be copied inline into the loop statement code, allowing concurrent execution of the loop and reducing the overhead required for a subroutine call.

A variety of techniques are developed which allow the compiler to generate parallel code for loop statements which contain data dependencies. Some of these are described in [Babb,1984], [Brode,1981], [Macke,1986], [Muller,1976], and [Padua,1986], and will be briefly described here.

A common sequential programming practice is to use variables for temporary storage in order to speed up program execution. Consider the following loop:

```fortran
DO 30 I = 1,1000
   TMP = I + 3 - J
   A(TMP) = B(TMP) + 3 * I
30   CONTINUE
```

In this example, the variable TMP is used in order to avoid calculating the common array subscript twice for each iteration. A data dependency analysis would reveal an output dependency and antidependency between each loop iteration, preventing the generation of parallel code. Inspection of the usage of TMP, however, indicates that the variable is not used to maintain data between loop iterations. Therefore, a separate TMP variable can be declared for each loop iteration. This is done, of course, by transforming TMP into an array, with one array element for
each loop iteration:

```
DO 30 I = 1,1000
  TMP(I) = I + 3 - J
  A(TMP(I)) = B(TMP(I)) + 3 * I
30 CONTINUE
```

No dependencies exist now between loop iterations. Therefore, the compiler can generate code to execute each loop iteration independently.

Another useful technique is the splitting of a single loop into two or more separate loops. Consider the following loop statement:

```
DO 30 I = 2,1000
  A(I) = A(I) + B(I)
  B(I) = A(I-1) + B(I)
30 CONTINUE
```

This loop's iterations cannot be executed concurrently since each iteration requires a result, stored in the array A, from the previous iteration. The loop can be split, however, as shown here:

```
DO 30 I = 2,1000
  A(I) = A(I) + B(I)
30 CONTINUE
DO 40 I = 2,1000
  B(I) = A(I-1) + B(I)
40 CONTINUE
```

Each of these loops can now be executed in parallel.

Many loop statements accumulate a sum, with each iteration modifying the total sum. The following is an example:
Although data dependencies exist between each loop iteration, a sophisticated compiler would recognize that partial sums could be accumulated for subsets of the loop iterations. One way to partition the above loop statement is a technique known as strip mining. In strip mining, a simple loop, such as the one above, is transformed into two nested loops. The above loop could be transformed as follows:

```
DO 40 J = 0,900,100
    SUM(J/100) = 0
DO 30 I = 1 TO 100
    SUM(J/100) = SUM(J/100) + A(I+J) + B(I+J)
30 CONTINUE
40 CONTINUE
TOTALSUM = 0
DO 50 I = 1,10
    TOTALSUM = TOTALSUM + SUM(I)
50 CONTINUE
```

In this transformation, a single loop of one thousand iterations is converted to a loop of one hundred iterations nested inside a loop of ten iterations. Each inner loop accumulates an independent sum. Therefore, there are no data dependencies between iterations of the outer loop, allowing each inner loop to be assigned to a separate processor in a parallel computing machine. Finally, a small loop of ten iterations is used to accumulate the sums, generating a total sum equivalent to the result obtained from a sequential execution of the same program.
A similar technique can be applied to a loop statement which accumulates a product. As with a sum, a product can be broken down into partitions, with each partition calculated concurrently. As before, the individual products would then have to be multiplied to calculate the correct value.

Another technique used to eliminate unnecessary data dependencies is to interchange the ordering of nested loops. Consider the following nested loop statement:

\[
\begin{align*}
&\text{DO 10 } J = 1,1000 \\
&\text{DO 20 } I = 1,1000 \\
&\quad X(I,J) = Y(I,J) + X(I-1,J) \\
&\quad 20 \text{ CONTINUE} \\
&\quad 10 \text{ CONTINUE}
\end{align*}
\]

The inner loop contains a flow dependency, since each iteration must wait for the result produced by the previous iteration. Therefore, this statement requires that the inner loop be executed sequentially. Notice, however, that the outer loop contains no dependencies. Furthermore, the ordering of these nested loops can be exchanged, producing the following:

\[
\begin{align*}
&\text{DO 20 } I = 1,1000 \\
&\text{DO 10 } J = 1,1000 \\
&\quad X(I,J) = Y(I,J) + X(I-1,J) \\
&\quad 10 \text{ CONTINUE} \\
&\quad 20 \text{ CONTINUE}
\end{align*}
\]

This statement will produce results identical to the previous statement. Additionally, the inner loop now contains no data dependencies between loop
iterations. Therefore, the individual iterations of the inner loop of this transformed statement can be executed concurrently on a parallel processing machine.

Despite the variety of techniques available for removing data dependencies and allowing the generation of parallel code, parallelizing compilers are often unsuccessful at generating efficient parallel code, and therefore must obtain specific information from the programmer.

Many Fortran translators, such as KAP/ST-100 described in [Macke, 1986], allow the programmer to specify which loops are suitable for concurrent execution. This specification is accomplished using directives within the program. After performing a translation on the original source code, the compiler produces output indicating which program segments were converted into parallel code. The programmer can then examine the portions of the program which were not converted to parallel code, and indicate to the compiler if changes should be made.

One common situation which requires user intervention is loops with complicated expressions for array indices. Often, the user can specify, for example, that the value of a certain expression will always be within specified bounds. This type of information can enable the compiler, in certain situations, to generate parallel code for loops too complex to transform automatically.

User directives can also be used to prevent the generation of parallel code. For example, in a previous example, accumulated sums and products were partitioned so that partial results were independently determined, with the final
results accumulated. For some algorithms, however, this type of translation can introduce discrepancies between the parallel and sequential versions of the program due to floating point round-off errors. Therefore, in some instances, the user may wish to tell the compiler to generate sequential code for certain loops.

PTOOL, described in [Allan, 1986], provides a user-friendly interactive environment for converting sequential programs into equivalent parallel programs. PTOOL first translates the sequential program in a standard manner, expanding parallel code where possible. Then, PTOOL steps through the program interactively, asking the user questions about loops which could not be expanded. The user can then specify that the loop iterations can be executed concurrently. Additionally, the user can alter the original program in order to allow parallel execution. Throughout the process, PTOOL provides helpful information regarding the reasons parallel code could or could not be generated.

Refined Fortran, described in [Dietz, 1986], is typical of a class of parallel versions of sequential languages in which the base language is extended to include facilities to control a parallel machine. Common language extensions include the FORALL, DOACROSS, and DOALL statements, which specify that loop iterations or array accesses should be performed concurrently. Other extensions include methods of requesting access to shared variables, allowing them to be modified in a controlled manner.

As described in [Dietz, 1986], extended versions of sequential language, where the user specifies the concurrencies, are more successful at generating
efficient code. This reflects the fact that the programmer is able to make more accurate decisions regarding independent subroutines and loop iterations. However, this approach does not permit the programmer to use a familiar standard sequential language. Instead, new programming techniques and methods must be learned.

2.1.1.2 Parallel Processing Granularity

As mentioned earlier, the data dependence analysis techniques described in the previous section are used in different types of compilers which produce code according to different models of parallel execution. A convenient way to classify these different compilers is to consider the granularity of the generated code.

Granularity, as described in [Babb,1984], [Gajski,1982], and [Kruatrachue,1988], is a measure of the amount of computation performed by a processor in a parallel machine before communication or synchronization with another processor must take place. Although there are no widely accepted standards regarding granularity size for a particular parallel model, most programs can be classified as having a large grain size or a small grain size.

In the previous section, much attention was given to the concurrent execution of individual loop iterations or groups of iterations. Some discussion of parallel execution of independent subroutines was also presented. These types of parallel program execution are classified as having a large grain size. Another approach to generating parallel code involves breaking down arithmetic equations
into independent sub-equations and executing each portion on a separate processor. This type of evaluation involves each processor doing a very small amount of processing between each data communication, and therefore is considered to have a small grain size.

Large grain sequential to parallel translation schemes are described in [Babb,1984], [Brode,1981], [Dietz,1986], [Kruskal,1985], [Macke,1986], [Padua,1986], [Allan,1986], and [Tuomenoksa,1985]. The large grain model of execution is advantageous because the system spends more time executing and less time communicating and synchronizing. This allows for greater efficiency and, therefore, faster program execution. The chief disadvantage with compiler translated large grain parallel programs is that it is very difficult for a compiler to recognize many potential concurrencies. Sequential programs can be very complex, and the compiler often has no information, for example, regarding the values of variables which are input by the user or read from a file. Therefore, these compilers are often unsuccessful at generating efficient parallel code. One solution to this problem is to require the programmer to either modify the original program in order to eliminate data dependencies or specify program statements which can be correctly executed concurrently.

Small grain compilation techniques are presented in [Brent,1973], [Moiske,1985], and [Muller,1976]. An advantage of this scheme is that the instructions decomposed for parallel evaluation are much simpler than those for large grain parallelism. Therefore, the compiler is better able to determine correct
partitions of the source code without extensive user interaction. Small grain parallel code is much less efficient than large grain code, due to the overhead incurred by frequent communication. This is particularly true for multiple processor systems with high speed processing capability but comparatively slow communication resources. Many existing parallel machines fit this description.

In response to this situation, a special class of parallel computation known as data flow is an area of great interest. Data flow computer techniques are discussed in [Bohm,1985], [Dennis,1980], [Gajski,1982], [Gostelow,1980], and [Milewski,1985]. The processing elements in a data flow computer, known as operation units, are not general purpose microprocessors, as in a general purpose MIMD machine. Instead, each is a simple device which accepts input operands, performs a simple operation on this data, and produces output operands. Intermediate data results are stored in an activity store. A fetch unit retrieves operands from the activity store and presents them to an operation unit. An update unit receives output operands from the operations units and stores them in the activity store. An instruction queue controls the activities of the system components.

Special data flow programming languages, such as Id, LAU, and Val, are developed which have features well suited to the data flow computer architecture. [Gajski,1982] discusses the principle features of these languages.

One characteristic of data flow languages is that variables represent values, not memory locations as they do in imperative languages such as Fortran, Pascal,
or C. Therefore, advanced techniques for memory allocation and garbage collection are used to automatically map variables into appropriate memory locations and de-allocate memory, transparent to the programmer.

Many data flow languages use a single assignment rule. This rule prevents a variable from being assigned different values during a program's execution, effectively eliminating the possibility of output dependencies and antidependencies. Single assignment also eliminates complex side effects which can prevent a compiler from converting standard sequential programs into equivalent parallel implementations.

[Gajski, 1982] claims that implicit parallelism, where the compiler extracts the parallelism from a sequential specification, is insufficient for powerful programming of a parallel machine. He states that typical programs, with spurious flow dependencies, can only be efficiently translated into parallel code if the programmer explicitly specifies concurrent operations.

2.1.2 User Specified Parallelism

Many programming languages have been designed or modified in order to give the programmer the ability to explicitly express parallel algorithms. The following is a discussion of the primary parallel features of a representative cross section of these languages.

2.1.2.1 Occam
Occam, described in [Inmos,1988] and [May,1987], is a parallel language designed for use with the Inmos Transputer. The Transputer is a microprocessor intended for inclusion in parallel processing machines. The instruction set and hardware of this processor facilitate process creation, process execution, inter-process communication, process synchronization, message passing, and multi-tasking.

Occam utilizes two basic low-level instruction statement types: assignment and communication. Assignment statements are similar to assignments in sequential languages. Communication statements read or write variables to or from channels; the channel is a data type of Occam. Reads and writes are synchronous and unbuffered. Therefore, a process writing to a channel will be blocked until a corresponding process performs a read from the same channel. Additionally, a process reading from a channel will be blocked until another process writes to the channel. Therefore, channels can be used for both communication of data and multiple process synchronization.

Occam combines the above simple statements into a variety of structured statements. The first type of structured statement is the SEQ statement. A SEQ statement executes each following statement sequentially, similar to a standard sequential language. The list of statements to be executed sequentially as part of the SEQ statement is indented; the end of the indented block of instructions indicates the end of the SEQ statement. Each statement within the SEQ statement can be a simple statement or a structured statement, such as another SEQ
Another type of structured statement is the PAR statement, which causes several indented statements to be executed concurrently. The PAR statement is terminated when all enclosed statements are terminated. The same rules of indentation apply to the PAR statement as to the SEQ statement. Each of the statements within the PAR statement, for example, could be a SEQ statement, corresponding to groups of sequential statements which are to be executed in parallel. Similarly, a group of PAR statements could be grouped together in a SEQ statement, causing groups of statements to be executed in parallel, one group after another.

The ALT statement is used to execute certain instructions based upon the arbitrary selection of a channel which has been written to by another process and is ready to be read from. The ALT statement consists of a list of guards; a guard is a read from channel statement. Corresponding to each guard is a statement. This statement can be a simple statement or a structured statement, such as a SEQ, PAR, or another ALT. When the ALT statement is executed, each of the guards is inspected in order to find which are ready for communication. One of the statements corresponding to a ready guard is arbitrarily selected and executed. If no guards are ready, the ALT statement performs no action.

[Kallstrom, 1988] compares a Transputer implementation of Occam to other representative parallel languages on different machines. The chief criticism of Occam is that it is restricted by the Transputer implementation. The Transputer
processor has four hardware communication links. The test program used in [Kallstrom, 1988] had multiple processes on each Transputer. Due to implementation restrictions, all processes on each Transputer could only have a total of four channels open at any given time for inter-process communication. Additionally, each process could only communicate with processes located on directly connected Transputers. Therefore, in order to send a message to a remote Transputer in the machine, processes must be written for each intermediate processor to buffer the message and relay it on the correct path.

Another criticism raised against Occam is that the programmer must manually assign each parallel process onto a physical processor. This is not only laborious, but also results in programs optimized for a particular network, composed of a specific number of processors interconnected in a certain configuration.

### 2.1.2.2 Concurrent Pascal

[Brinch Hansen, 1977] and [Hartmann, 1977] describe Concurrent Pascal, an extension of the sequential language Pascal used to write concurrent programs. Brinch Hansen designed Concurrent Pascal to write concurrent programs for operating system applications. He implemented the Solo operating system in the language.

Concurrent Pascal uses the process and monitor data types to create concurrent programs. A process is an executable program entity. It contains local
data and instructions to operate upon this data. A process cannot access variables which are not local to the process. Therefore, processes cannot directly access global variables. A process is created and activated using an init operation. Once created, a process exists forever. A process is always executing, unless it is waiting to execute within a monitor.

A monitor is a data type which, like a process, contains local data and instructions to operate on the local data. These instructions, however, are grouped into entry procedures. Multiple processes can execute an entry procedure within a monitor. Therefore, processes can use monitors to communicate or synchronize. For example, in order to communicate, process A could call an entry procedure PUT in a monitor M. The procedure PUT would receive a parameter from A and store the value of the parameter in a local variable of M. Process B could then call an entry procedure GET in monitor M. The procedure GET would copy the local variable written to by PUT into a parameter which would be returned to process B. In this manner, a data value is transferred from process A to process B.

Although multiple processes can execute entry procedures within a monitor, Concurrent Pascal does not allow more than one process to execute within any given monitor. If a process A attempts to execute within a monitor that another process B is currently executing within, process A will be suspended by the operating system until process B exits the monitor. This exclusive access to the monitor provides for data integrity.

Consider a monitor which accumulates a sum. Multiple processes send
partial sums to the monitor using an entry procedure. The code of the entry procedure reads a local variable which contains the total sum, adds the partial sum to this value, and stores the result back into the local variable which represents the total. With exclusive access, this monitor will function properly. Without it, however, the possibility of inaccurate data exists, since two concurrent processes could try to read and modify the same variable at the same time.

Monitors are useful in operating system software for handling hardware resources, such as printers, disk drives, and terminals. In a multi-tasking system, multiple tasks will attempt to use these resources concurrently. The monitor, with its exclusive access, can guarantee that, for example, only one task will use the printer at any given time.

Concurrent Pascal is unique in that the programmer must specify, at compile time, which processes have access to specific monitors. In this manner, the compiler is able to signal access errors. In addition to aiding the debugging process, this aspect of Concurrent Pascal enables more efficient run time code, since access rights need not be checked after the program is compiled.

2.1.2.3 C Language Extensions for the Intel iPSC

[Kallstrom,1988] describes an implementation of the C language, augmented with special functions to facilitate process creation, inter-process communication, and host interaction. This implementation is designed for the Intel iPSC hypercube architecture parallel machine. A similar C implementation
exists for the NCUBE parallel processing computer, and is described in [Hayes,1986]. Unless otherwise specified, the following discussion applies to both of these language extensions.

These extended C language implementations require the programmer to develop a sequential process for each processor. Special functions from a C library are used by the independent programs to accommodate the needs of parallel programming.

The functions send() and recv() are used to send and receive messages from other nodes. These are nonblocking functions, meaning that they return immediately even if no message is prepared. The functions sendw() and recvw() perform the same task, but these are blocking commands; they will not return until a corresponding process performs a send or receive. Each of these functions require the process to supply parameters specifying the physical processor number and process identification number of the receiving process, the communication channel number, the type of the message, the location of the message, and the length of the message.

Other functions provided in these systems are cubedimO, mypidO, and mynode(). The function cubedimO returns an integer which specifies the dimension of the hybercube. The number represents the base two log of the number of processing nodes. The function mypidO returns a logical number which represents the calling process. Similarly, mynodeO represents the physical number of the processor on which the calling process is executing.
These system configuration functions allow programs, at run time, to determine the characteristics of the machine architecture. Programs could therefore be written to adapt to the machine size. For example, consider a program which performs a repetitive calculation for each element of a very large array. Each process executing the program could call cubedim() to determine the number of processors in the system. The total array size would then be divided by this number, yielding the number of elements each processor should operate upon. Each process would then call mynode() to determine which physical processor it is located on. This number could then be used to select an appropriate portion of the array to act upon.

While this scheme does promote transportability to larger hypercube machines, the programmer is now forced to write a program which contains both the algorithm of the application as well as system routines to control the run time behavior of the program. This latter need is better served by the operating system.

A function syslog() allows processing nodes to write a message to a log file for later analysis of a program's run time behavior. This feature is very useful for both debugging and optimizing parallel programs, since timing information for communicating processes indicates a great deal about a parallel program's behavior. It is also useful for locating the causes of deadlocks. Unfortunately, calls to syslog() slow the execution dramatically, so the inclusion of them may alter the inter-process synchronization timing.

On the Intel iSPC system, a process management program must be written
to execute on the host machine. A special library of C functions are provided to facilitate this task.

The host communicates with the processing nodes in order to provide user input and output as well as disk file usage. The commands to perform this are similar to those executed on the individual nodes. The functions sendmsg() and recvmsg() are used to send messages to and receive messages from an individual processing node. Each of these require parameters to specify the communication channel number, the type of message, the address of the message buffer, the length of the message in the buffer, the node identification number of the processor to receive the message, and the process ID of the corresponding process. Additionally, the sendmsg() function returns as a parameter the number of bytes actually received.

The host uses the load() function to load the executable code for a program onto a processing node and begin execution. The lwaitall() function is used to cause the host to wait until a node process terminates. The lkill() function is used to abort the program running on a particular processing node. Any of these functions can be used on a single node or on all nodes.

Writing an application for the Intel system therefore requires two separate programs: one for the host machine, and one for the parallel processing nodes. These two programs are linked at compile time to produce a executable program.

2.1.2.4 Ada
Ada is described in [Baker, 1985], [Bjorner, 1980], [Cherry, 1984], [Helmbold, 1985], [Stratfore-Collins, 1982], and [Gothe, 1991]. Like Occam, Ada is a language originally designed to permit the expression of parallel algorithms.

Ada presents a new data type, called a task. A task is an executable program entity. They are similar in nature to a procedure or a subroutine in that they consist of program statements as well as local variables for data storage. Unlike procedures, however, tasks can be executed concurrently with other tasks.

Tasks are defined within a program block. They can be local to a procedure, much as local types, constants, and variables are permitted. Additionally, tasks can be declared as part of the main program block.

Tasks are automatically invoked when the block they are declared in becomes activated. For example, if a task is declared within a procedure, that task begins execution when that procedure begins execution. Additionally, that procedure will not complete until all declared tasks are completed. If a task is declared in the main block of a program, the task will be invoked when the program is started. The program will not be terminated until all tasks declared in the main program block are terminated.

The distinction between tasks and procedures is significant. Procedures are executed only when they are explicitly called by another procedure or the main program. When a procedure is called, the calling program is suspended until the procedure terminates. Through the use of control statements, such as an if clause, a procedure need not be called if the parent procedure chooses not to. Tasks, on
the other hand, are implicitly activated when the parent procedure is executed. It executes concurrently with the procedure which activated it. A task is always executed when its parent is executed.

The basic task model serves to synchronize the activity of several tasks. All tasks are activated at the same time. No other tasks will be executed until all have terminated. Often, however, tasks need to synchronize their execution during the execution of the task, not merely at the beginning or ending of the task. Examples of this are accessing shared variables or the passing of data from one task to another. Ada uses the rendezvous to accomplish this synchronization.

A rendezvous is a common point for synchronization in two tasks. A rendezvous is the action of two tasks, N and M, for example. N must call an entry in M. Concurrently, M must execute an accept statement which corresponds to N's entry call. When these two criteria are met, the two tasks are synchronized. During this period of synchronization, M can execute one or more program statements. These statements can include the assignment of a value, passed as a parameter by N in its entry call, into a local variable of M. In this manner, communication can take place between tasks. Alternatively, task M could, during the period of synchronization, read or write a value to a shared global variable, with the knowledge that N could not access that variable during the rendezvous. A further option would be for M to execute no statements during the rendezvous; such a rendezvous would serve only as a synchronization. After M completes the statements within its accept, both tasks can continue executing the next sequential
instruction in their respective task bodies.

In the above rendezvous description, the term task can also apply to the procedure or main program statement that implicitly activated another task. A common Ada practice is to consider this procedure or main block as a master task. This is an accurate description, since this block executes concurrently with all its declared tasks.

[Gehani,1988] is a comparison of the Ada and Concurrent C parallel programming facilities. For several reasons, Gehani considers some aspects of the Ada constructs to be inferior to those of Concurrent C.

One problem cited discussed by Gehani is the single direction transfer of data in the Ada rendezvous. The situation is rectified in the recently accepted extended rendezvous. Another of Gehani's criticisms is that Ada does not allow parameters to be passed to tasks. In order to do so, either an extra rendezvous must be used, or parameters must be hard-coded into the task definition. Finally, Gehani considers the automatic activation of Ada tasks to be a disadvantage; a more suitable method is the explicit creation and termination of these parallel entities.

2.1.2.5 Modula 2

Modula 2, described in [Beidler,1986], [Christian,1986], and [Hernandez,1992], is a programming language created by Niklaus Wirth, the creator of Pascal. Although Modula 2 is very similar in nature to Pascal, it
contains several new programming facilities. Among these is the coroutine, a construct used to express parallel programs.

A coroutine is a sequentially executed program entity which can operate concurrently with other coroutines. Coroutines are created and activated using the types PROC and PROCESS and the procedures NEWPROCESS and TRANSFER. The type PROC allows a variable representation of a procedure. When using coroutines, a procedure is referenced as a parameter of type PROC. This reference causes the coroutine to execute the instructions associated with that procedure.

The type PROCESS is merely a pointer to a storage area in memory. This storage area is used to maintain state information for the coroutine.

The procedure NEWPROCESS is used to allocate memory space for a new coroutine. Four parameters are passed to NEWPROCESS. The first is of type PROC; it indicates the parameterless procedure which will be the code for the coroutine. The second and third parameters represent the location and size, respectively, of data and variables to be used by the coroutine. The final parameter is of type PROCESS. This parameter is returned by NEWPROCESS, and is merely a pointer to a data area used by TRANSFER.

The procedure TRANSFER is used to activate a coroutine. Two parameters are passed to this procedure. Both are of type PROCESS. The first process parameter represents the current process, while the second represents the coroutine to be started. TRANSFER turns control of the system over to the coroutine. The coroutine then returns control back to the main program with a
subsequent TRANSFER call.

All implementations of Modula 2 include a library called Processes which exports a type and several procedures intended for further flexibility in implementing coroutines. The type SIGNAL is used for synchronization between multiple coroutines. It is similar in concept to the semaphore. The procedure Init is used to initialize a variable of type SIGNAL. The procedure WAIT is used by a coroutine which is waiting to synchronize with another coroutine. WAIT causes a coroutine to enter a waiting state until another coroutine executes a SEND procedure. The single parameter for the SEND and WAIT procedures is of type SIGNAL. The procedure Awaited returns a boolean value of true if at least one coroutine is in a waiting state for the indicated SIGNAL variable. Otherwise, it returns a value of false. Finally, the procedure StartProcess is used to start one or more coroutines for concurrent execution.

Modula 2 is an attractive choice for parallel programming in that all concurrent directives are system independent. Unfortunately, programming in Modula 2 requires user specification of many parameters, such as the amount of memory required to store a coroutine’s state variables. These values are system dependent, and therefore require modification if a parallel program is to be implemented on a different parallel machine.

2.1.2.6 LC3

LC3 is an extension of the sequential language C. The principle aim of LC3
is experimentation with different schemes for process communication and synchronization. It is described in [Banatre, 1985].

LC3 extends the C language with a parallel kernel. This kernel consists of objects and main operations. The different types of objects are the process, the sequence, and a family of atomic actions. The main operations are the write sequence, the read sequence, and process activation.

A process is an executable program entity which can execute concurrently with other processes. Processes may have local data. Additionally, processes may have child processes. Arrays of processes can be declared if the same executable structure is to be replicated. Additionally, linked lists of processes can be created. The linked list is well suited to programs where the processing requirements are not known at compile time. A linked list can grow dynamically, allowing the program to expand the list, and hence create additional processes, at program run time.

A sequence is an ordered list of data. The structure of a sequence is similar to that of a queue. Sequences are created by writing successive values into the sequence. After all data is written, the sequence is closed. In order to retrieve data from a sequence, a process must use a channel to access the sequence in a controlled manner. Reads from a sequence extract single data values in the same order that they were written to the sequence. A process reading an empty sequence will be suspended at run time until data is written into the sequence.

Atomic actions are sequences of instructions during which, while being
executed, the executing process performs no communication with other processes. A family of atomic actions exists when a group of processes each is executing a related atomic action. The processes executing within the family can interact with one another, but none may interact with processes not within the family. A family of atomic actions is used to synchronize multiple processes in a similar manner as the rendezvous of Ada. Unlike the rendezvous, however, any number of processes can execute within a family of atomic actions.

2.1.2.7 Concurrent C

Concurrent C, described in [Gehani, 1988] and [Gehani, 1990], is another extension of the sequential language C. The methods of expressing parallelism, however, are unlike those of other extensions of C described previously.

Concurrent C is described as being a higher level language than C language extensions which consist of function libraries to facilitate inter-process communication. Like Ada and Modula 2, Concurrent C incorporates data types for parallel process creation, and new control structures to coordinate and synchronize to activities of these entities.

In many regards, Concurrent C is similar to Ada. The process type of Concurrent C is similar in concept to the task of Ada. Unlike Ada tasks, however, Concurrent C processes are explicitly created and activated, rather than coexisting with the program block that declared them. Also, Concurrent C allows parameters to be passed to processes. This enables a more efficient invocation of processes,
since a rendezvous is not needed to send parameter values as is done in Ada.

Concurrent C employs the rendezvous mechanism for inter-process communication and synchronization, and its usage is very similar to the rendezvous of Ada. In Concurrent C, however, the extended rendezvous is available, which allows bidirectional data communication in a single rendezvous.

In both Ada and Concurrent C, multiple tasks can make an entry call to the same entry in a given task. Concurrent C offers more flexibility than Ada in enabling the accepting task to select one of the tasks executing the entry requests.

2.1.2.8 Linda

Linda is not actually a programming language, but rather a set of parallel programming tools which can be used to extend a sequential language. Linda is described in [Gelemter,1985], [Gelemter,1988], and [Leler,1990]. Current implementations of Linda are based on the sequential languages Fortran and C.

The primary entity of a Linda system is referred to as "tuple space," or TS. An implementation of Linda consists of a TS coupled with several operations which act upon TS. TS is used to store tuples, which are sequences of typed fields. An example tuple is (3.14159,'A',true).

Four commands are used to operate on the TS. The command out(t) adds the tuple t to TS. The command in(u) removes from TS a tuple, t, that matches the template specified in u. The actual values of the fields of t are assigned to the formal parameters in u. The command rd(u) is identical to in(u), except that the
tuple is copied into \( u \) but not removed from TS. If a process executes a \( \text{rd}(u) \) or \( \text{in}(u) \) operation and no tuple in TS matches \( u \), the process will be blocked until an appropriate tuple is placed in the TS.

The final operation used with the TS is \( \text{eval}(t) \). This operation is similar to \( \text{out}(t) \), in that the tuple \( t \) is moved into the TS. The difference is that \( t \) is not evaluated until it is placed into TS. This type of tuple is called a "live tuple." Live tuples become executing entities which execute in parallel with the process that placed them in the TS. After a live tuple terminates its execution, its results are copied into the TS. It can then be read out by another process.

A parallel program is executed by placing two or more live tuples into the TS using the \( \text{eval}(t) \) operation. Each of these live tuples can execute \( \text{out}(t) \) to communicate. Alternatively, live tuples can create additional live tuples. Communications using the TS can be used to synchronize multiple processes.

Linda implementations are well suited to large information systems which access database resources. One current application implemented with Linda is a DNA database program. This program first places many live tuples into the TS. These live tuples are used to search for DNA sequence matches. The program then writes a large database called GenBank into TS; GenBank is a list of known DNA sequences. Each entry of GenBank is written as a tuple. Each live tuple continually removes a GenBank sequence from TS, compares it to the search string, and places a tuple into TS indicating its success or failure. The master program continually removes these result tuples from TS until a successful match
Currently, Linda is implemented on a network of autonomous VAX/VMS machines.

2.1.2.9 CSP

The language Communicating Sequential Processes, or CSP, is described in [Hoare,1978]. This is a classic paper, in which Hoare develops the necessary mechanisms to control a multi-processor machine using multiple sequential processes. These processes communicate and synchronize using matched communication commands.

CSP uses a parallel command to simultaneously activate multiple concurrent processes. Each of these processes is a sequential series of program statements. A process can access only its own local variables; there are no global shared variables. The parallel command terminates when all activated processes terminate.

Input and output commands are used to pass data values between processes. Each input command must specify the process which will execute a matching output. Each output command must specify the corresponding process performing a input command. In addition to transfer data, input and output commands perform process synchronization. A process performing an output command will be suspended until the input command in the named process is executed. Therefore, no buffering is performed on the communicated data. CSP programs
can use dummy communications, where no meaningful data is transferred, to synchronize processes which have no need for communication.

CSP uses input commands as guards in a mechanism for nondeterministically selecting from multiple processes which could concurrently perform output commands. This is useful for implementing a client-server algorithm, where various client processes randomly issue requests to a common server. The server process needs to determine which, if any, processes are requesting service. The server would use input statements as guards to select an appropriate service request.

A guarded command is selected for execution if the input command corresponds to an output command that is currently ready for evaluation. If multiple input statement guards are ready, one will be arbitrarily chosen for execution. When an input guard is recognized as ready and selected, the input command is executed. Other statements associated with that guard are then executed. In this manner, a ready input is arbitrarily selected, the input command is performed, and other program statements relative to the input command are executed.

Input statement guards can also be used in repetitive commands. In this structure, the command repeats until all processes named in the input statement guards have terminated. This also is useful for client-server algorithms. By using a repetitive statement with input command guards, the server process could accept client requests in an arbitrary ordering. After all perspective client processes are
terminated, the server process could also terminate.

CSP provides a simple template matching scheme which enables a process to select input communications which have data corresponding to a specific pattern. This enables a process to discriminate between different types of communicated data and allows, for example, prioritization of process activities.

2.1.2.10 Parallel Pascal

Parallel Pascal is designed for use with the Massively Parallel Processor, or MPP, developed by Goodyear Aerospace for NASA. It is described in [Reeves, 1985]. Parallel Pascal introduces several constructs to express parallel computations on the MPP. The MPP is a parallel array processor. Therefore, Parallel Pascal's features strongly reflect the array architecture of the MPP.

Parallel Pascal allows all standard expressions, as used in conventional Pascal, to be evaluated over entire arrays using the parallel directive. For example, in conventional Pascal, the scaler variables A and B could be added together, with the sum stored in a scaler variable C. In Parallel Pascal, this same operation is performed on A, B, and C, where these variables are now arrays, rather than scalers.

Array selection is another feature of Parallel Pascal. Two dimensional arrays can be referenced, as in standard Pascal, by referring to the row and column indices of the desired elements. In Parallel Pascal, where portions of arrays can be accessed, two dimensional arrays can be referenced by specifying index ranges.
In this manner, a submatrix can be modified, inserted, or extracted from a two dimensional array.

Parallel Pascal includes several array reduction operations. These perform tasks such as calculating the sum of all elements in an array, the product of all elements in an array, the logical ANDing or ORing of all array elements, or the determination of the maximum or minimum value in the array. The array to be reduced by these operations can be a submatrix if so specified.

Parallel Pascal supports conditional evaluation for each array element through the use of the WHERE statement. A where statement specifies a boolean operation to be performed on each individual array element. If the boolean result evaluates to true, a specified operation is performed on the array element. Otherwise, a default operation is performed on the element. This statement is equivalent to the IF statement of standard Pascal, except that the boolean expression is evaluated for each individual array element, and the result impacts the operation on that element independent of other array elements.

2.2 Operating Systems

Operating systems play an important role in the evolution of computer systems [Peterson, 1985]. Early computers required the human operator to perform the tasks performed today by an operating system. These tasks included manually loading a program, using console switches, punched cards, or paper tape, pushing the appropriate buttons to initialize the computer and run the program,
monitoring the program using indicator lights on the console, and receiving output on punched cards or paper tape.

Today's operating systems automate all of the above tasks. Additionally, modern operating systems facilitate disk file accesses, time sharing, file sharing, interactivity, multi-tasking, networking, computer security, and multi-processing. Clearly, the success of today's modern, complex computers is largely due to the advancements made in operating system theory. Future advancements in operating systems will be necessary in order to fully utilize the resources of parallel processing machines [Hwang, 1982], [Li, 1987], [Lin, 1987].

This section of the literature review will discuss operating system developments. Particular emphasis will be given to topics related specifically to parallel processing, concurrent programming, and to the operating system kernel developed in this dissertation.

The first provisions within operating systems for multiple processor systems were not for the parallel processing of a single application. Instead, the first use of these machines was for the execution of multiple independent programs simultaneously in order to achieve greater throughput. No provisions were made for parallel processing requirements such as shared variable accesses, inter-process communication, or process synchronization. Operating system developments in this area concentrated on scheduling algorithms, resource sharing, and job partitioning, all of which are pertinent to parallel processing.

[Gabbyow, 1982] describes an algorithm for scheduling multiple independent
processing jobs on a system with two processors. The algorithm is designed such that total execution time is minimized. [Gonzalez, 1978] considers the problem of partitioning M independent jobs onto a system of N parallel processors. Both of these approaches require that jobs be ordered by the length of required execution time. The algorithms improve throughput by running short jobs first, saving long jobs to run later when few short jobs exist. [Peterson, 1985] demonstrates that running short jobs first is optimal since the waiting time for a given group of jobs is minimized. Peterson shows that scheduling a short job before a long job decreases the wait time of the short job more than it increases the wait time of the long job.

The problem with ordering jobs according to execution time is that there is no way to accurately determine the length of a job. One approach used in batch systems is for the programmer to manually specify the approximate execution time by assigning each job a class. Short jobs are assigned a higher classification than long jobs, and therefore are given priority. User assigned classifications are generally not accurate; furthermore, errors in assigning the class of a job can cause serious errors within the scheduler, resulting in poor system performance.

The Solo operating system is described in [Brinch Hansen, 1977]. Solo is a single-user operating system. A variety of independent, concurrent processes are utilized to control accesses to system resources such as input and output devices, memory, file systems, and inter-process communications. It is designed for use with the PDP-11/45 computer system. The Solo system is almost entirely written
in a high level language, Concurrent Pascal. Two people were able to write all the code for the Solo system in less than one year.

The Unix operating system, described in [Ritchie, 1974], is one of the most popular operating system in use today. Unix was developed at Bell Laboratories as a research project. Several different implementations of Unix exist; the differences between these, however, are irrelevant to the topics to be discussed for this dissertation project. Unix is one of the first operating systems to provide mechanisms for concurrent application processing.

The first version of Unix was implemented on a PDP-7 computer and written in assembly language. Subsequent versions were largely rewritten in the C programming language, which was developed at Bell Laboratories to support Unix. The use of a portable high level language to implement Unix contributes greatly to its popularity; versions of Unix are available for many different computer systems, with most system and application software compatible with most machines.

Unix is an interactive, time-sharing operating system. It is written in so that the user interface is performed by a simple program that can be modified to fit the individual needs of the user. A tree structured file directory is used in Unix. All types of input and output, using such devices as disk files, printers, and terminals, is performed in a consistent manner, allowing the input of output of a program to be redirected to and from various devices. Additionally, the output of one program can be treated as the input of another concurrent program, allowing
multiple pipeline processes to execute simultaneously.

Unix supports parallel programming through the use of processes. Each executing program is considered by Unix to be a process. Any currently executing process can activate a new process by using the fork command. A process which creates a new process is called a parent; a newly created process is called a child. A child process can become a parent by creating an additional new process. After a parent creates a child using fork, both the parent and the child continue executing the instructions following the fork. Fork returns a result parameter to the child with a value of zero. A non-zero result is returned to the parent; this value is the integer used to identify the newly created child process.

Immediately after the fork, both the child and the parent are executing within the local memory space of the parent. Typically, a child process will execute a system routine to create a new unique address space. The child can then use local variables in the new address space without corrupting the values of variables in the parent's address space.

A child process terminates by executing an exit command. A parent process can wait for a child to terminate by executing a wait command. The wait command returns a result parameter which is the integer identifier of the terminated child process. This used to determine which child process terminated if multiple child processes were activated.

A primitive mechanism called a pipe is provided for inter-process communication in Unix. A pipe is a queue structure, with data added to the pipe
by one process and removed by another in a first in first out manner. A pipe has a finite size, typically a few thousand bytes. A process attempting to remove data from an empty pipe or a process attempting to add data to a full pipe is suspended by the operating system.

More advanced versions of Unix also incorporate the socket mechanism, which is similar to a pipe. The difference is that pipes are generally local to a single processor, whereas sockets allow communication between processes executing on separate processors. Sockets are therefore utilized extensively in Unix networks. There are several different classes of sockets, each of which function in a slightly different manner. A stream socket is a reliable communication link which transports data in variable sized groups of bytes. A sequenced packet is similar to a stream socket, except that record boundaries are utilized. A datagram socket carries variable length data messages, but the messages do not always arrive at the destination process in the same order that they left the source process. Also, there is no guarantee that a datagram message will arrive. A reliably delivered message socket is similar to a datagram socket, except that messages are guaranteed to arrive at the destination process. Finally, a raw socket is a low level communication path which has no communication protocol. All other types of sockets use a raw socket to perform the actual data transfer. User programs can use a raw socket to implement a new socket protocol.

Standard Unix possesses some characteristics which are difficult to implement on a multi-processor machine. One such characteristic is the use of
processor priority raising during event indication in order to prevent race conditions. It is not possible to block other processors in a shared memory machine by using this technique.

Nearly all Unix implementations provide some form of networking support. This includes supports for passing messages between machines and exchanging mail or news files. No facilities are provided, however, for remote activation of processes with parameters or for distributed file systems. These features, if needed, must be implemented as a user program. [Presotto,1990] describes an extension to the Unix operating system for cooperative processing between unrelated processes. The approach utilizes naming conventions that facilitate high level inter-process communication services.

[Hwang,1982] describes a job scheduling system implemented within a Unix environment. This system incorporates a kernel routine which maintains the state of each machine within a network. Stored within the state information is an indication of the load each processor is currently executing. The system works by requesting each processor to provide information regarding its current processing load. When a new job is created, it is assigned to the processor with the lightest load. This system is static, in that the determination of the location of a job is determined before the job is executed. After the job becomes activated, it remains on its assigned processor until completion.

[Black,1990] describes programming techniques used to provide hints to the operating system regarding scheduling priorities. These techniques require the
programmer to invoke library functions that instruct the operating system to
deschedule the current and resume execution of another specified task.

[Lin,1987] describes a technique for dynamic job processor relocation. Lin
states that, for large multi-processor computer systems, a load balancing
mechanism based upon centralized control is not practical. Lin's approach is based
upon a distributed algorithm executed on all processors in the system. The
algorithm involves idle or under utilized processors sending requests to adjacent
processors. These requests are for additional work. If one of the adjacent
processors is busy with several jobs, it will respond to the request of the idle
processor by relocating one or more of its currently executing jobs onto the idle
processor. This process is repeated at regular time intervals. The system is able
to gradually spread jobs evenly throughout the computer network.

[Ross,1991] presents an optimal scheduling algorithm for a multi-processor
system. The emphasis here is on the development of two separate scheduling
mechanisms: a global process placement algorithm, and a local processor job
scheduling scheme. Two different classes of processes are considered in this paper.
Generic jobs are jobs that can execute on any processor. Dedicated jobs must be
executed on a specific processor. The local scheduling algorithm gives priority to
dedicated jobs.

[Upfal,1987] addresses the concept of shared variables on a distributed
memory system. In a shared memory parallel processing machine, each processor
has access to a common variable storage space. This space is frequently used to
store shared variables which can be read or written, in a controlled or uncontrolled manner, by any processor in the system. Distributed memory systems, on the other hand, are characterized by a lack of shared memory. Upfal describes a method for simulating shared variables on a distributed memory machine by maintaining multiple copies of shared variables in the local memories of multiple processors. Whenever a write operation is performed on one of these copies of a shared variable, all other copies must be either invalidated or changed to reflect the action of the write. Upfal shows that suitable performance can be obtained in a distributed system if only a majority of the copies of altered shared variables are updated. An algorithm is presented which determines which copies of the variables are to be updated in order to minimize communication distance within the network for distributing and using the modified variables.

[Tanenbaum,1985] discusses several issues considered in the design and implementation of distributed operating systems. Particular emphasis is given in this paper to the dissimilarities of distributed operating systems and computer networks.

One issue Tanenbaum discusses is the remote activation of a procedure on a separate processor within the multi-processor system. A critical problem arises when dealing with parameter and function passage between the processors. Standard sequential languages pass parameters in two ways: by value and by reference. With a value parameter, the actual value of the parameter is copied into the address space of the procedure. This can be readily accomplished in a
distributed system by passing the parameter value as a message over the interconnection network.

A reference parameter in a single processor system is passed as a pointer to the actual variable data. This mechanism allows a procedure to change the value of the variable. Unfortunately, reference parameters cannot be easily passed between processors with distinct address spaces, since a pointer is only valid within the memory of the processor which defines it. Structured system type variables, such as files, can be accessed by multiple processors within a machine by utilizing a system-wide pointer mechanism. Simple variables, however, would create a great deal of system overhead if a global system pointer were created for each. Therefore, problems remain for an efficient implementation of returning parameters from remote procedures.

Another design issue discussed by Tanenbaum is the algorithm to be used for scheduling concurrent processes onto parallel processors. One scheduling algorithm discussed is based on co-scheduling. Co-scheduling involves placing all processes which inter-communicate on a different processor. Multiple groups of communicating processes are placed on the same group of parallel processors, with the result that each processor executes multiple processes, but no communicating processes are located upon the same processor. At run time, a co-scheduling scheme executes a processes within a group concurrently. This is accomplished by having each processor within the system first execute processes in a single group. When the time slicing interval terminates, each processor in the system would
deschedule the current process and begin executing a process from the next group. In this manner, all processes that must communicate will be executing concurrently; therefore, communication delay times will be reduced since a process executing a send need not wait for a corresponding receiving process to be scheduled.

Several problems arise from co-scheduling. First, a hardware mechanism must be incorporated in order to efficiently synchronize the time slicing of each processor. Often, processes within a group are waiting upon one process to issue a command. During this time, all other processes within the group are idle. A co-scheduled system would force a large percentage of the system to be idle, while processes from other groups could utilize the processor time efficiently. Finally, groups of processes inter-communicate with one another. By placing each process on a distinct processor, all communication must take place between processors. For most parallel processing systems, processes which exist on the same processor are able to communicate more efficiently.

In comparison to the developments in parallel programming languages, relatively little work exists in the area of parallel operating systems. One reason for this is that existing programming systems for parallel processing, such as those described in [Kallstrom,1988] and [Hayes,1986], function using an operating system similar in nature to a single processor system. In these systems, most tasks required for parallel process execution are performed by the user within the user's source coded program utilizing calls to a small number of predefined subroutines.
2.3 Summary of Literature Review

Many positive attributes exist in several existing parallel processing programming systems. The language Occam is notable for its introduction of language structures to express sequential and parallel program entities. Occam also contains a sophisticated process communication mechanism for message passing.

Concurrent Pascal and Ada introduce the concepts of the Process and the Task, respectively, which are used to create concurrent program entities. Concurrent Pascal also introduces the Monitor, which provides for controlled, structured accesses to critical code regions and shared variables.

It is clear that currently available tools for parallel programming are not well suited to the efficient development and maintenance of parallel programs. The reasons for this include the following:

The programmer is responsible for not only partitioning the program into parallel entities, but also determining a suitable physical machine assignment for each parallel entity. While manual partitioning is shown to be advantageous, manual machine assignment is difficult, requires program modification if the target machine is expanded or reduced, and results in programs which are machine specific or architecture specific.

Existing languages require the programmer to code many of the low-level aspects of process creation, process termination, inter-process communication, shared memory allocation, and critical code execution. These are tasks which
clearly should be the responsibility of the operating system.

With some notable exceptions, current parallel programming languages do not utilize high level control structures to initiate and synchronize concurrent program entities. High level control structures simplify program development, facilitate debugging, and increase program clarity, and are therefore desirable.

Many existing parallel languages are designed for specific machines or machine architectures. In many situations, this is a desirable attribute. For example, Occam contains many functions, as well as some restrictions, which are particular to the Inmos Transputer processors for which Occam is designed. Through the use of Occam, efficient programs can be developed which make use of the Transputer processor's multi-tasking, synchronization, and communication capabilities. Unfortunately, these same functions cannot be performed as efficiently on other processors or architectures.

No generic method is developed which permits controlled simultaneous accesses to shared variables by cooperating concurrent tasks. The monitor of Concurrent Pascal guarantees exclusive access to a variable; this approach, however, is not suitable for the common situation where multiple processes are reading from a common variable, such as within an image processing algorithm.

Of the existing parallel languages surveyed, only Concurrent C provides the facility to pass parameters to a process. In most other languages, an additional communication, rendezvous, or monitor call must be performed in order to perform the equivalent action of a parameter initialization. None of the existing
languages provide a mechanism for returning parameters from a process located on a disjoint processor without explicitly performing a communication.

[Pancake, 1990] discusses the attributes of existing parallel processing programming languages as related to the needs of scientific programmers. The paper states that existing languages are well suited for experimentation with the non-determinant aspects of parallel processing. Most scientific programmers, however, do not wish to deal with non-determinacy but rather want to code efficient programs. The paper concludes that these programmers do not have appropriate tools for this task.

[Skillicorn, 1990] proposes a computational programming model that is applicable to a broad class of parallel computer architectures. This model is based upon the Bird-Meertens formalism, the application of an algebra to develop a set of applied computational theories. The resulting language model can be applied to SIMD machines, tightly coupled MIMD machines, hypercube interconnected machines, and grid interconnected machines.

2.4 Goals of Dissertation Project

The goal of this dissertation is to develop a programming system for parallel processing. This system utilizes features of existing sequential and parallel languages which have been found to be useful and effective. Additionally, new features are developed in an attempt to correct some of the shortcomings of existing systems.
The parallel programming language developed is an extension of an existing programming language. This is beneficial in that programmers do not have to learn an entirely new language. With this language, the programmer has the responsibility of explicitly specifying the concurrent components of the program.

The language incorporates control structures and data types necessary to concisely express parallel programs. These are incorporated into the existing language in such a way that the style of programming is not altered dramatically. Specifically, the new control structures and data types permit the definition and execution of parallel processes, controlled inter-process communication, exclusive accesses to critical code regions, and protected concurrent accesses to shared variables.

This language is implemented on the Pyramid Processing System developed at The Ohio State University, as described in [Chen, 1989]. However, the language is not designed specifically for this machine. A primary goal of this project is the development of a system that can be used to program a variety of different parallel processing computing machines. Therefore, the language attempts to provide generic mechanisms for performing the tasks necessary for efficient, controlled parallel processing.

A primary criticism of existing parallel programming languages is the requirement that the programmer specify the machine allocation to be used for the program's execution. Additionally, the programmer is required to hard code such tasks as shared variable accesses, inter-process communications, process
invocations, and process terminations. In order to avoid this situation, this parallel processing programming system includes an operating system which automates these tasks. This operating system operates transparently to the user, operating at a level below that of the programming language. Compiled programs written in the language contain calls to the operating system to perform such tasks as inter-process communication, process creation, process synchronization, and shared variable accesses. These tasks automate such decisions as processor allocation for tasks, inter-process communication physical address resolution, terminated process identification, critical code execution, and shared memory allocation and usage.

The operating system is designed such that decisions regarding machine allocation are made at execution time. These decisions are made based upon two sets of data: First, the machine constantly keeps track of the usage of memory, processor busy time, and communication channel busy time for each processing element within the system. This information is useful for placing processes within the system into locations which have the lightest work load. Second, the language is designed in such a way that all program entities to be accessed by a concurrent process must either be local to the process or be passed to the process as a parameter. By examining the parameters passed to each process, the operating system determines which processes need to communicate or synchronize with one another. Based upon this information, the operating system attempts to place related processors near one another within the processor network in order to minimize communication distance and, therefore, reduce program execution time.
In addition to performing tasks specifically to facilitate parallel processing, the operating system also implements such standard tasks as program loading, terminal input and output, and file input and output.

In order to demonstrate the operation of the parallel processing programming system, several demonstration programs are written and executed on the Pyramid Processor System. The operating system task of automated process placement is demonstrated by presenting several application programs. This is accompanied by a discussion of the parallel algorithms used in each of the application programs, with emphasis on the relationships between concurrent processes.

In order to demonstrate a valuable role of the operating system, application programs are executed on several different configurations of the Pyramid Processing System. The system is reconfigured by selectively enabling and disabling processors. Reducing the number of active processors increases the amount of time needed to execute the applications. However, the applications do not require alteration for the different system configurations. The execution times for these configurations are presented.

2.5 Project Contribution

This project represents some unique contributions in the area of parallel processing software systems. The system allows parallel processing programs to be implemented without an intimate knowledge of the target machine architecture.
The BPL language uses structured control statements to express parallel algorithms.

The operating system automates the task of processor allocation, a task that is usually performed manually by the programmer. The programming language compiler generates "hints" to the operating system, indicating which processes will communicate with one another. The operating system uses this information when determining processor allocation. [Austin, 1991] states that "Few of the load-balancing schemes described in the literature deal with the implications of load-balancing on communication." Therefore, this work is unique and significant.

The software system is designed to facilitate hardware scalability; unchanged programs can execute if processors are added or removed from the system. This is accomplished by the BPL language, which allows the user to program for a "virtual" parallel processing machine, and by the BOS operating system, which performs a processor allocation that is a function of the number of processors in the system and of the relationships between concurrent processes in the program.
Chapter III

Language Specification

This chapter is a detailed discussion of the BPL programming language. BPL is an extension of the sequential programming language Pascal, described in [Jensen, 1974]. The extensions consist of data types and control structures which facilitate the expression of parallel algorithms. This chapter will define all extensions BPL makes to standard Pascal. The sample programs in Chapter VI contain practical examples of the usage of these extensions.

In addition to the parallel programming constructs, the BPL language contains other extensions which are useful for performing various low-level machine access operations. These extensions were necessary to facilitate the writing of the BOS operating system. A section of this chapter describes these language extensions.

A variety of standard languages have been augmented in order to permit the expression of parallel algorithms. In addition to Pascal, parallel implementations exist of C and Fortran. Languages which permit concurrent programming, such as Ada and Modula II, also have been modified and used for parallel programming experimentation. The final section of this chapter discusses the choice of Pascal
as the base language for BPL.

A simplified Backus Naur Form, or BNF, is used to define the syntax of language components. All non-terminals are enclosed in less-than and greater-than symbols, such as `<procedure heading>`. All terminals are underlined. Each BNF expression consists of a left hand side, an equality symbol ( == ), and a right hand side. The left hand side is a non-terminal. The right hand side consists of non-terminals and terminals joined in combinations using operators.

The | operator represents an OR function. An expression of the form `<a> | <b>` represents the production `<a>` or the production `<b>`. Optional items appear in square brackets. The expression `<a> [ <b> ]` represents the production `<a>` or the production `<a> <b>`. If an asterisk appears following the right bracket, the optional item can be repeated any number of times, including zero. The expression `<a> [ <b> ]*` can represent productions such as `<a>`, `<a> <b>`, `<a> <b> <b>`, or `<a> <b> <b> <b>.

3.1 BPL Data Types

Data types in BPL are much more diverse than those of standard Pascal. In Pascal, data types are used only to describe the manner in which variable data is stored and accessed in system memory. BPL data types, in addition to specifying the storage format of variable data, also specify sequences of program statements which define the operations to be performed on this data. This encapsulation of data and operations to be performed on the data allows the programmer to specify
concurrent entities which can be distributed on a parallel processing machine. Data types in BPL also permit concurrent processes to communicate, synchronize, and share common variables.

This section defines the data types of BPL. These data types can be used anywhere standard Pascal data types are used. For example, BPL types can be used in type declarations, in variable declarations, or in other type declarations, such as fields of a record or elements of an array.

3.1.1 The Process

In BPL, a process is a data type which defines an entity which can be executed simultaneously with other processes. Processes are defined in the type declaration portion of a program, procedure, function, or even another process. Furthermore, processes can be used as components of other types, allowing, for example, an array of processes. The form of a process type definition is shown below

\[ <\text{process type}> \equiv \text{process} \; [\text{<process parm list>}] \; ; \; \text{<declarations>} \]

\[ \text{<body>} \]

The <declarations> are identical to declarations in Pascal, and can include constants, types, variables, procedures, and functions. The <body> also is identical to the program body of Pascal. The <process parm list>, however, is different from the parameter list of standard Pascal. It, and its non-terminals, are
defined as follows:

\[ \text{process parm list} = = \{ \text{parm} \ [ \text{parm} \ ]^* \} \]

\[ \text{parm} = = \text{val parm} \mid \text{var parm} \mid \text{return parm} \]

\[ \text{val parm} = = \text{id name} \ [ \text{id name} \ ]^* : \text{type} \]

\[ \text{var parm} = = \text{var id name} \ [ \text{id name} \ ]^* : \text{type} \]

\[ \text{return parm} = = \text{return id name} \ [ \text{id name} \ ]^* : \text{type} \]

Val parameters are parameters whose value is sent to the process prior to the execution of the process. Var parameters are parameters whose value is sent to the process prior to the execution of the process and returned to the parent following execution of the process. Ret parameters are parameters whose value is returned to the parent following execution of the process. Val, var, and ret parameters may be considered as input, input/output, and output values, respectively.

Like procedures in Pascal, variables local to a process exist only while the process is executing. When the process is terminated, the variables are discarded. If a process is invoked repeatedly, its variables are created and discarded for each execution; no values are stored between executions.

A process can only reference procedures and functions declared within the process. Constants and types declared in any outer block, as defined in standard Pascal, can be referenced in a process. Variables declared outside a process can only be accessed within the process only if they are passed as a val, var, or ret
parameter to the process.

Processes are executed using a parallel control statement, discussed in section 3.2.1 The Parallel Statement. A process can only be executed by the program or process which declares the process. Processes can be nested to any level.

3.1.2 The Resource

A resource is similar to a process in that it encapsulates both data and the instructions for accessing that data. Unlike processes, however, resources contain variables which exist during the lifetime of the program or process which declares the resource. Also, resources can have several different procedures available to the calling program to access the internal data, while a process behaves like a single procedure.

A process can only be executed by the process in which it is declared. Resources, on the other hand, can be passed as val parameters to other processes and resources, allowing them to be executed by any process or resource which is an ancestor of the process in which the resource is declared.

Resources are the only type of executable entity which can be executed by more than one process. For this reason, resources are useful for data exchanges. Any process that executes a procedure within the resource can examine or modify the contents of its local variables. Because of this property, resources are implemented in a manner so that only one process can execute a procedure within
a resource at any given time. When a process is executing within a resource, the resource is said to be busy. When a process attempts to execute a procedure within a resource, the operating system first determines if the resource is busy. If the resource is not busy, the process is allowed to execute within the resource. If the resource is busy, the process is added to a busy queue. When a process exits a resource, it checks to see if any other processes are waiting for use of the resource. If so, the next process on the queue is allowed to execute within the resource. In this manner, processes are granted access to the resource on a first-come-first-serve basis.

The BNF representation of a resource declaration is shown below:

```
<resource type> == resource interface <interface> implementation
<implementation> end
```

The resource declaration is composed of two sections: the interface and the implementation. The interface describes the names of the procedures within the resource which can be accessed by an external process. Also included here are the names and types of all parameters for these procedures.

```
@interface> == <proc header> [ ; <proc header> ]*

<proc header> == procedure <id name> [ <parm list> ]
```

The <parm list> in the <proc header> is identical to a process parameter list; each parameter is notated as val, var, or ret, corresponding to input, input/output,
or output parameters, respectively.

The implementation section of a resource declaration contains the declarations of all variables local to the resource. Also included within the implementation section are the bodies of the procedures whose headers appear in the interface section. Furthermore, other procedures, functions, types, and constants may be included in the implementation section.

\[
\text{<implementation> } = = \text{ <imp entry> } [ ; \text{ <imp entry> } ]^* \\
\text{<imp entry> } = = \text{ <declaration> } \mid \text{ <interface proc body> }
\]

\[
\text{<interface proc body> } = = \text{ procedure } \text{id name} ; \text{ <declarations> }
\]

All procedures whose names appear in the interface section must have a corresponding procedure body in the implementation section. Note, however, that the parameter names are not repeated with the procedure body; they are only specified in the interface section.

A resource type declaration does not actually create a resource which can be executed. In order to do this, a variable must be declared whose type is the specified resource type. Resources can be components of other structured types; for example, it is possible to create arrays of resources, or to create records with one or more fields which are resources.

In order to execute a specific procedure within a resource, the programmer must specify both the name of the resource variable and the name of the resource procedure to be executed. The syntax for this is as follows:
\texttt{<resource exec> == <resource var name> \&<resource proc name> <resource proc parameters>}

Note that the \texttt{<resource var name>} could be, for example, an array variable name with an index specifier or a record variable name with a field specifier.

Resources have essentially the same restrictions with regard to accessing objects external to the resource. A resource can only reference procedures and functions declared within the resource. Constants and types declared in any outer block, as defined in standard Pascal, can be referenced in a resource. Variables declared outside a resource can only be accessed within the resource if they are passed as a \texttt{val}, \texttt{var}, or \texttt{ret} parameter.

Resources can be passed as parameters to processes. They must, however, be passed as a \texttt{val} parameter.

3.1.3 The Channel

Channels are used in BPL for communication and synchronization between processes. They have a syntax and usage form which is similar to that of typed files in modern Pascal. A channel declaration has the following form:

\texttt{<channel type> == channel of <type declaration>}

Channels are used in the same manner as files in Pascal. The Write operation is used to send data on a channel. The Read operation is used to receive data from a channel. Channels are unidirectional. A sending process opens a channel using
A Rewrite command. A receiving process opens a channel using a Reset command. Use of a channel is terminated using a Close operation. The syntax for each of these operations is shown below:

\[
\text{<write op> } = = \text{ write } \langle \text{<channel id>}, \text{<var id>} \rangle \\
\text{<read op> } = = \text{ read } \langle \text{<channel id>}, \text{<var id>} \rangle \\
\text{<rewrite op> } = = \text{ rewrite } \langle \text{<channel id>} \rangle \\
\text{<reset op> } = = \text{ reset } \langle \text{<channel id>} \rangle \\
\text{<close op> } = = \text{ close } \langle \text{<channel id>}, \text{<var id>} \rangle
\]

As can be seen, the syntax for most of these operations is identical to that used in modern Pascal for using typed files. The only exception is the close operation. Close actually performs a write operation; it then closes the channel. This is done so that the close event is properly synchronized between the processes at each end of the channel.

BPL provides a boolean function which can be used by a receiving process to determine if a sending process has closed a channel. This function is called eoc, which stands for "End Of Channel." It is analogous to the "End Of File" function of Pascal. The form of eoc is

\[
\text{<eoc expr> } = = \text{ eoc } \langle \text{<channel id>} \rangle
\]

Eoc returns a true value if the named channel has been closed by the sending process. Otherwise, it returns a value of false. The eoc function is useful for
processing an arbitrary amount of data. The receiving process can continually test the value of eoc. If it is false, the receiving process can read data from the channel; if it is true, the receiving process can assume that no more data exists.

Channel operations are synchronized in BPL. When a process performs a write operation on a channel, that process will wait until another process performs a receive operation on the same channel. Similarly, a process performing a rewrite operation to initialize a channel for sending will be suspended until another process performs a reset operation to initiate the same channel for receiving. A close operation enforces the same synchronization as a write operation. The process performing the close will wait for a corresponding read operation. The read which corresponds to a close operation behaves identically to a read corresponding to a write operation. The only difference is that eoc becomes true for the read channel after performing a read operation which corresponds to a close.

Because reads and writes are synchronous, a process cannot communicate with itself using channels; to do so, the process would have to simultaneously perform read and write operations. The control structures used to invoke processes do not permit a process to execute concurrently with a child process. (For more information, refer to section 3.2 BPL Control Structures.) Therefore, a process cannot communicate with a child process using channels.

From the above discussion, the following rules regarding the use of channels are derived: A channel must be declared in a parent process. The parent
process will pass the channel as a val parameter such that two processes receive the channel. Of these two processes, neither can be a direct descendent of the other.

The following examples illustrate the correct use of channels:

Process A declares a channel, K. Processes B and C are children of process A. Process A passes channel K as a val parameter to both processes B and C. Processes B and C communicate using channel K, with one process acting as a sender and the other process acting as a receiver.

Process A declares a channel, K. Process B is a child of process A. Processes C and D are children of process B. Process A passes channel K as a val parameter to process B. Process B passes channel K as a val parameter to both processes C and D. Processes C and D communicate using channel K, with one process acting as a sender and the other process acting as a receiver.

Process A declares a channel, K. Processes B and C are children of process A. Process D is a child of process B. Process E is a child of process C. Process A passes channel K as a val parameter to both processes B and C. Process B passes channel K as a val parameter to process D. Process C passes channel K as a val parameter to process E. Processes D and E communicate using channel K, with one process acting as a sender and the other process acting as a receiver.

At first glance, the usage of channels appears quite cumbersome. For example, it might be easier if a process could reference a channel declared in any ancestor process rather than requiring all channels to be passed as parameters to processes. The reason channels must be passed to processes is that, in doing so, the operating system is able to determine which processes will be communicating with one another. By using this information, the operating system can determine an efficient distribution of processes within the parallel processing system.
3.1.4 Shared Variables and Templates

BPL allows processes to share variables in a controlled manner through the use of shared and template data types. Shared data types are declared as follows:

```
<shared type dec> == shared <type declaration>
```

Shared variables represent data which multiple processes can read or write simultaneously. Parameters passed to processes are actually copies of variables which exist in the parent process. After completion of the process, var and ret parameters are copied back into the parent process. Shared variables are not copies of data, but actual dynamic variables which are globally altered during process execution, not at the completion of a process.

Obviously, allowing all processes to modify shared variables could result in erroneous data if two processes attempted to modify a common variable simultaneously. Therefore, accesses to shared variables must be restricted such that they are performed in a safe manner. Templates are used to provide this restricted access.

Shared variables cannot be directly read or written. Instead, requests are made to read or write some or all of a particular shared variable. Access is then made to the shared variable using a template variable, which performs the necessary address mapping. After the process is finished with the shared variable being referenced through the template, a release operation is performed.

A template declaration has the following form:
The <type declaration> of the template specifies the type of shared variable which can be accessed through the template. For example, if the shared variable to be accessed is of type integer, the template used to access the shared variable must also be of type integer.

In order to use a template, the process must first issue a request to use a specific shared variable. Two types of requests are permitted: read requests and write requests. Write requests allow the process to read or write to the shared variable, while a read request only allows reading the shared variable. The requests are defined as follows:

\[
<\text{read request stat}> = \text{ReadRequest} \left( <\text{shared variable name}> , <\text{template name}> , \left[ <\text{index range selector}> \right] \right)
\]

\[
<\text{write request stat}> = \text{WriteRequest} \left( <\text{shared variable name}> , <\text{template name}> , \left[ <\text{index range selector}> \right] \right)
\]

The <index range selector> is used if and only if the shared variable is an array type. It allows a process to request a portion of a global array. It has the following form:

\[
<\text{index range selector}> = [ <\text{index range}> [ , <\text{index range}> ]^* ]
\]

\[
<\text{index range}> = <\text{integer}> . <\text{integer}> | *
\]

Therefore, a process could, for example, request a range of rows and columns within a larger array. This area could correspond to a neighborhood of pixels.
within an image or a sub-matrix. If an asterisk (*) is used instead of an integer range specification, the entire row or column of the array is selected. If the element type of a shared variable requested for a read or write is an array type, an index range must be specified for each dimension of the array.

If a process requests a shared variable which is currently unavailable, the process will be suspended by the operating system. This is transparent to the programmer. After issuing a request for access to a shared variable, the process may access the shared variable using the template, substituting the name of the template for the name of the shared variable. The shared variable name is only used when requesting access; after access is granted, the template serves as an alias to the requested shared variable. No other references to shared variables are permitted.

After a process is finished with the shared variable it is accessing, the release operation is used to relinquish the variables currently locked by the template, with the following syntax:

```
<release stat> == Release ( <template name> )
```

Templates are released automatically if a new request is made for the same template or when a process containing templates terminates. All requests and releases are handled by the operating system.

Shared variables can be passed as a val parameter to a process or resource procedure. Templates, however, are local to a process or resource procedure, and
therefore cannot be passed as a process or resource procedure parameter. Shared variables and template variables can, however, be passed to standard procedures and functions as var parameters.

BPL creates a resource for each shared variable. All requests and releases are processed by internal calls to this resource. When a process issues a read request, the resource checks to see if any other process is currently writing to the requested shared variable. If no other process is currently granted write access to the requested shared variable, the request is granted. If another process is currently granted write access to the requested shared variable, the process issuing the read request will be suspended by the operating system until the shared variable is available.

When a process issues a write request, the resource checks to see if any other process has read or write access granted to the requested shared variable. If not, the request is satisfied. If, however, another process is currently accessing the requested shared variable, the process is suspended until the shared variable is available. Additionally, all further read or write requests to that shared variable will be suspended, allowing the write request to eventually be granted.

It is the programmer's responsibility to ensure that deadlock does not occur due to improper use of shared variables and templates. Consider the following scenario:

Process A first issues a write request for shared variable X. The request is granted. Process B then issues a write request for shared variable Y. This request is granted. Next, process A issues a read request for shared variable
Y. This request cannot be granted, since process B has write access to Y; therefore, process A is suspended until Y becomes available. Finally, process B issues a read request for shared variable X. This request cannot be granted, since process A has write access to X; therefore, process B is suspended until X becomes available. Both processes A and B are deadlocked, since each is waiting on the other to release a shared variable, but neither is able to do so.

One goal of the BPL language project is independence of machine architecture. The shared and template data types are designed such that they can be implemented on a shared memory machine or on a network of inter-connected processors with local memories. The current implementation of BPL is for a shared memory system. Therefore, the shared and template data types are implemented using the shared memory. Template accesses are performed by assigning to each template an address range. The memory management unit for the processor is then modified so that the template address forces an access to the shared memory at the address of the actual shared variable.

In a network inter-connected distributed-memory parallel processing machine, the same mechanism for requesting and releasing shared variable accesses is utilized. The actual method for accessing the shared variable through the template, however, must be modified since no shared memory mechanism exists.

Shared variable accesses are accomplished by distributing the shared variables throughout the network of processors. When a program begins its execution, no shared variables would exist. In order for a shared variable to be created, a write request must be issued for it. At this time, the requested shared
variable would be created and stored in the memory of the processor that requested it. After releasing the newly created shared variable, it would remain in the memory of the processor that created it. Multiple processes can be granted read access to the same shared variable simultaneously by distributing multiple copies of the shared variable throughout the system.

When release operations are performed, the copies of the distributed shared variables can remain in the local memories of multiple processors. When a process is granted write access to a shared variable, however, all other copies of the shared variable must be discarded, as they are no longer valid. If a particular processor runs out of memory space, it can discard shared variables stored there which are no longer being used. At any time, however, at least one copy of any shared variable must be maintained within the local memory of a processor.

The resource which controls accesses to shared variables in a distributed memory system must therefore perform more functions than the corresponding resource in a shared memory system. These additional functions include:

1. Keeping track of which processors have copies of each shared variable. This must be done in order to prevent multiple copies of the same shared variable from being stored in any processor’s memory.

2. Discarding all extra copies of any shared variable which has been granted to a process for write access.

3. Maintaining at least one copy of each shared variable at all times.

It must be noted that, in a distributed memory system, programs which
utilize shared variables extensively may not execute as efficiently as they would on a shared memory system. This is particularly true if shared variables are frequently written to and then read by many processes; this would require frequently copying data throughout the network. Shared variables can, however, assist with the implementation of programs with large arrays of numerical data on processor networks. They allow arrays to be declared which are larger than the local memory of any single processor. They also relieve the programmer of the task of determining where each row or column of an array is currently being stored; the operating system handles this task automatically.

3.2 BPL Control Structures

The BPL data structures provide facilities for specifying entities for concurrent processing, resource management, communication, and synchronization. In order to use these data structures, they must be incorporated into programs using the control structures which BPL provides. Each of these control structures comprises a statement, and may therefore be used anywhere in BPL that a standard statement is used in Pascal.

3.2.1 The Parallel Statement

Three steps must be performed in order to create and execute a process. First, the process type must be declared. This is described in section 3.1.1 The Process. Second, a variable must be declared of the process type defined in step
one. (Note that, as in standard Pascal, the process type can be defined in the type declaration section or in the variable declaration section.) Third, the process variable must be executed within a parallel statement.

The parallel statement is used to execute multiple processes concurrently. The syntax is as follows:

\[
\text{<parallel stat> } = \parallel \text{ <process stat>} \ [ ; \text{ <process stat>} ]^* \text{ end}
\]

A \text{<process stat>} indicates the name of each process to be executed concurrently, along with the names of the parameters to be used with each process. The syntax for a \text{<process stat>} is as follows:

\[
\text{<process stat> } = \text{ <process var id>} \ [ \text{ <parm list>} ]
\]

\[
\begin{align*}
\text{for } \text{<index var id> } & := \text{<expr>} \text{ to } \text{<expr>} \text{ do } \\
\text{<process stat>}
\end{align*}
\]

\[
\text{<parm list> } = ( \text{<parm id>} \ [ , \text{<parm id>} ]^* )
\]

The for statement form of the \text{<process stat>} is used to execute each element in an array of processes. This saves the programmer from having to explicitly code the execution of each individual process. Also, the index variable can be used to select different parameters to be used with each element of an array of processes. For example, an array of resources might be declared, with one resource within the array corresponding to each process in an array of processes. By using the for statement, each element of the resource array can be selected as a parameter for each element of the process array. Another use of the index variable is to provide
an identification number for each process in an array of processes. To do this, the actual value of the index is passed as a val parameter to the process. The process can then refer to this parameter in order to determine its identification number.

Note that the recursive nature of the for statement allows several for statements to be nested. This feature allows multi-dimensional arrays of processes to be executed.

The optional parameter list is used only if the process type of the process variable includes parameters in its declaration. The type of each parameter must match the type of the parameter as it is declared in the process type declaration. Expressions are permitted for val parameters only; variables must be used for var or ret parameters. The value of the index variable in a for statement can be used in any expression for a val parameter. Additionally, the index variable can be used as an index into an array, allowing individual array elements to be passed to different processes.

When executed, a parallel statement initiates each of the processes included within the statement. The process containing the parallel statement is then suspended until all processes included within the parallel statement have terminated. This allows the programmer to properly synchronize the activity of several processes. All processes within the parallel statement are started at the same time; no further processing occurs until all processes have finished.

3.2.2 The Select Statement
The select statement is used by a process to arbitrarily select a channel which is ready for communication and perform specific operations based upon which channel is selected. A channel is said to be ready for communication if the other process which has opened the channel is currently executing either a read or write operation. The syntax of the select statement is as follows:

```plaintext
<select stat> == select <select clause> [ <select clause> ]* [ <else clause> ] end

<select clause> == <channel id> : <statement> ;

<else clause> == else <statement> ;
```

All channels included in select clauses must be opened for reading or writing; failure to do so will result in a run-time error. When a select statement is executed, a list is created of all channels within the select clauses which are ready to communicate. One of the corresponding select clauses is arbitrarily selected, and the statement within that clause is executed. If none of the channels within the select clauses are ready to communicate, the statement within the optional else clause is executed; if no else clause is included, the select statement terminates without executing any statement.

The select statement is useful for creating processes which must service requests from a variety of other processes. The processes requesting service would communicate with the server process by writing to a channel. The server process would then receive the requests by including each channel in a select clause. After arbitrarily selecting a request, the server process would perform a read on the 
channel, corresponding to the write from the requesting process. The server process could then service the request, performing actions based upon which channel the request came from and what data was read from that channel.

3.3 Low Level Programming Language Extensions

The BPL language is intended for use as an application tool for parallel programming. In order to support the run time needs of the BPL program, a suitable operating system is needed. This operating system is written in BPL. An operating system must perform certain low level tasks, such as controlling hardware devices and handling interrupts. In order to conveniently perform these tasks, BPL contains extensions which facilitate low level programming.

3.3.1 Exception Procedures

An exception routine is a procedure written to handle interrupts, software trap instructions, and other special processor conditions such as bus error or coprocessor violation. Exception routines that handle interrupts or processor conditions must be careful not to alter the value of any registers. Exception routines that service software trap instructions often inspect the values in the registers and, optionally, return values to the calling program in the registers.

BPL facilitates the writing of exception routines by allowing a procedure to be designated as an exception procedure. The syntax of an exception procedure declaration heading is as follows:
An exception procedure generates special code that saves all processor registers on the stack upon entry to the procedure and restores all register values upon exiting the procedure. For an exception procedure that handles hardware interrupts, the procedure would have no parameters. The procedure could handle any pending hardware events using standard programming constructs, including calling procedures, modifying local or global variables, or reading and writing hardware registers. Of course, the programmer must beware of the consequences of changing any global variables. The entire processor state, however, will be properly restored at the completion of the exception procedure.

Exception procedures that service software trap instructions must have the ability to read and write to the registers. In order to facilitate this, the entry code to an exception procedure builds a new stack frame as though the processor registers were parameters to the routine. At the completion of the exception procedure, these "parameters" are removed from the stack and stored back into the appropriate registers. The following procedure heading shows the correct parameters for use in a software trap handling procedure:

```pascal
procedure Trap2( FormatVector : word;
    ProgCntr : longint;
    StatusReg:word;
    a7, a6, a5, a4, a3, a2, a1, a0,
    d7, d6, d5, d4, d3, d2, d1, d0 : longint);
Exception;
```
An exception procedure with this heading could inspect the values of all registers and modify them if necessary. In addition to servicing software trap instructions, this type of procedure heading is also useful for a bus error handling exception procedure. This is facilitated by the use of the FormatVector parameter which provides bus error information in 680x0 based systems.

In order to use exception procedures, the address of the procedure must be stored in the processor's exception vector table. BPL enables this through use of the SetExceptionVec procedure, which has the following syntax:

\[
\text{<set exception stat> } = = \text{SetExceptionVec ( <exception number> , @ \text{<exception proc name> } )}
\]

When using this procedure, the <exception number> parameter is the exception vector number. After calling this procedure, all exceptions of the designated number will be handled by the named procedure.

It should be noted that the facilities for writing and setting exception vector handling routines work only in the supervisor mode of processing.

3.3.2 Inline Machine Instructions

Frequently, the operating system must perform register operations in order to configure the microprocessor. For example, the operating system needs to disable interrupts when executing certain critical operations. In order to perform such tasks, the BPL compiler provides a very simple mechanism for including inline machine language instructions. The syntax for this mechanism is as follows:
When an inline statement is encountered in the source code, the BPL compiler simply inserts the integer constants into the executable code. Each integer constant represents one sixteen bit instruction word.

In order to use inline instructions, the programmer must hand assemble the assembly instructions into their corresponding machine language values. This obviously is too cumbersome for a large number of instructions. It is, however, very useful for performing a small number of assembly language instructions.

3.3.3 Absolute Variables

Through the use of absolute variables, BPL programs can access absolute memory addresses. This ability is useful for programming hardware devices. Absolute variables also allow two different variables to share the same memory location. The syntax of an absolute variable declaration is as follows:

<abs var dec> == <var name> : <type dec> absolute <absolute addr value>

<absolute addr value> == <integer constant> | <var name>

Two variables sharing the same memory location can be particularly useful if the two variables are of different, incompatible types. For example, in a floating point to ascii conversion routine, a variable of type real can be declared to share
the same memory location as an array of bytes. In this manner, a floating point can be assigned to the real variable, while the individual bytes of the real variable can be inspected using the array of bytes.

3.3.4 Memory Operations

BPL provides mechanisms for accessing system memory at a low level. The first such mechanism is the move statement, which has the following syntax:

\[ <\text{move stat}> == \text{move} ( <\text{var1 addr}> , <\text{var2 addr}> , <\text{byte count}> ) \]

The move statement is used to copy a block of memory from one address to a second address. In the above BNF expression, \(<\text{var1 addr}>\) represents the source variable, \(<\text{var2 addr}>\) represents the destination variable, and \(<\text{byte count}>\) is an integer expression representing the number of bytes to be copied.

The addr expression is used to determine the address of a variable and convert it to an integer value. The syntax is as follows:

\[ <\text{addr expr}> == \text{addr} ( <\text{var name}> ) \]

Another useful construct is the ptr variable reference. Its syntax is as follows:

\[ <\text{ptr reference}> == \text{ptr} \ <\text{type dec}> \ <\text{integer expr}> \]

The ptr reference is used to convert an integer expression into a variable
reference of the type specified by <type dec>. The integer expression can be obtained in any manner, including the use of an addr expression. For example, to store a value of zero into the third byte of a variable named "X", the following statement would be used:

\[ \text{ptr byte ( addr(X) + 2 ) := 0; } \]

### 3.4 Pascal as the Base Language for BPL

Several factors were considered when selecting the base language for BPL. These factors reflect a variety of issues, including ease of use, consistency of implementation, and flexibility.

A variety of sequential languages, such as Fortran, C, and Pascal, have been used as the base language for parallel programming languages. This approach offers ease of use for sequential programmers; rather than learn a brand new language, programmers can express parallel algorithms within the framework of a familiar language.

Fortran was rejected due to its lack of block structure. Block structure offers many advantages to sequential programming, including modular program form and ease of program development and maintenance. In parallel programming, block structure plays a further role, in that program blocks are used to express concurrent program components.

The sequential language C serves as the base language for several parallel
languages. Typical language extensions of C incorporate functions which facilitate concurrent processes and inter-process communication. C makes extensive use of pointers to reference parameters in function calls. Unfortunately, this scheme for parameter passing does not work for processes executing on processors with disjoint memories. Therefore, in order to use the process parameter facilities of BPL, significant changes would be required to the C language. These changes would diminish the advantages of using a familiar language. Another drawback of C is its loose type matching rules. Parallel programming and debugging is very complex; relaxed type matching restrictions enables a programmer to make simple mistakes which can be difficult to locate and fix. Therefore, strict type matching is a desirable attribute in a parallel language.

The languages Ada and Modula 2 include facilities for expressing concurrency, making them good candidates for a base language. These languages were rejected for comparatively minor reasons. Indeed, both of these languages are very similar to Pascal. Ada is a very complicated language, making a complete Ada implementation very difficult. An implementation of an Ada subset would be more practical; experienced Ada programmers, however, would be likely to miss language features omitted from the subset, contradicting the advantages of using a familiar language. Modula 2 is not a widely used language, and therefore does not possess the benefit of familiarity for a majority of programmers.

Pascal serves well as the BPL base language for a variety of reasons. Its block structure allows the language syntax to be easily extended to express
processes and resources. Strict type matching rules prevent the programmer from making simple mistakes. Pascal contains key words for declaring parameters of different classes; these key words can be extended to allow the three process parameter types to be specified. Finally, Pascal is a popular, widely used language, enabling many programmers to become familiar with BPL in a short time.

3.5 Comparison of BPL with Other Languages

This section is a comparison of the features of the BPL language and the primary features of other parallel programming languages. This comparison does not describe every existing parallel processing programming language. Instead, a representative group is chosen that incorporates the basic language components found in existing languages.

The Occam language uses a statement similar to the parallel statement of BPL to invoke concurrent processes. Occam also incorporates a channel structure for message passing. Occam, however, does not control accesses to shared variables, as BPL does through the use of resources. In fact, Occam allows concurrent processes to reference global variables. This programming model does not support the execution of concurrent processes on multiple processors. The programmer is therefore responsible for removing any global variable references from processes intended to execute on separate processors. The BPL language only permits processes to access local variables and parameters passed to the process.

The Occam language is designed for the Transputer microprocessor, which
has four communication links. The Occam programmer is responsible for allocating each of these links for each processor. All concurrent processes on a given Transputer must have no more than four active channels that communicate with other processors. The BPL language implements virtual channels. Any number of channels can be used in a program. The BOS operating system performs the task of allocating system resources that support channel operations. The operating system does not impose any limit on the number of active channels on each processor; conflicts are resolved by communication processes.

The Concurrent Pascal language contains a monitor data type, which is similar to the resource of BPL. Concurrent Pascal processes, however, are different in that they are always executing. In BPL, processes are executed only when a parallel statement is executed. The programmer can send parameters to processes using BPL. In Concurrent Pascal, a monitor must be used to pass information to a process. Concurrent Pascal has no equivalent of the BPL channel. Monitors are therefore used for inter-process communication.

Concurrent Pascal was primarily designed for multi-tasking programming on a single processor machine. Processes are therefore permitted to access global program variables. Processes that do so would not execute properly on a network inter-connected parallel processing machine.

Programming parallel processing machines using the C language in conjunction with a parallel processing function library is very different from programming in the BPL language. No new language control structures are used
to facilitate parallel programming. Instead, a C function library is used that includes functions for starting concurrent child processes, communicating between processes, and determining if child processes have completed. Existing C language parallel function libraries do not automate the task of machine allocation. Therefore, the programmer is required to specify the target processor for each processor. Typically, the processor number is a parameter for the function that invokes concurrent processes. In contrast, the BPL language places processes automatically, basing allocation decisions on the number of processors in the system, the work load of each processor, and the communication relationships between concurrent processes.

C library communication functions also require the processor number as a parameter. In order for a process to send a message to another concurrent process, the physical processor number must be coded. BPL channel operations are performed by the operating system, which maintains all physical machine attributes without requiring the programmer to hard code this information.

Programs written in the C language therefore are designed for a specific machine architecture and, typically, a fixed number of processors. The resulting programs must be substantially rewritten if a new machine is used or if processors are added to or removed from the system. BPL programs are not architecture specific and are not written for a known system configuration. The result is a higher level of parallel programming where many aspects of the physical machine are hidden from the programmer.
Ada and Concurrent C are two similar parallel processing programming languages. Both support the concept of a task, a concurrent program entity. In Ada, however, tasks are automatically started when the procedure they are declared in starts; Concurrent C allows the explicit start of tasks in a manner similar to the parallel statement of BPL.

Both languages use the rendezvous to facilitate task communication or synchronization. The rendezvous is useful for synchronizing two concurrent tasks. The BPL resource, however, can be used for synchronizing any number of tasks. Additionally, the resource provides more flexibility for parameter data communication.

The BPL channel provides a mechanism more convenient for inter-process communication. Additionally, the syntax for BPL processes, resources, and channels allows the compiler to generate "hints" to the operating system that can be used to facilitate the determination of machine allocation.

BPL programs are of a higher level than parallel programs written in many other languages. The programmer is not required to create a program for a specific machine architecture and configuration. As a result, program development is simplified, system scalability is facilitated, and efficient programs are created.

3.6 BPL Implementation of the Dining Philosophers Problem

This section describes a BPL implementation of the Dining Philosophers Program, a classic synchronization problem first described in [Dijkstra, 1965]. The
problem consists of a group of philosophers seated at a circular table. A bowl of rice is located in the center of the table. For each philosopher there is one chopstick placed on the table. The chopsticks are located between the philosophers. The philosophers perform two tasks: thinking and eating. In order to eat, a philosopher must pick up two chopsticks. These chopsticks are located on either side of the philosopher. Because of this arrangement, it is impossible for two adjacent philosophers to eat at the same time. When a philosopher is finished eating, he sets down the two chopsticks and resumes thinking.

The solution to this problem must prevent deadlock. Additionally, each philosopher must not be starved. The BPL implementation of this problem utilizes a resource type to handle the distribution of the chopsticks. Three entry procedures are provided for this resource. The first entry procedure, Initialize, is used to place all the chopsticks. The entry procedures PickUp and PutDown are used to obtain and return a pair of chopsticks, respectively.

The resource maintains an array of boolean variables to represent the state of each chopstick. Additionally, the resource maintains an array that represents an ordered list of the philosophers that have tried unsuccessfully to pick up a pair of chopsticks.

The PickUp procedure works as follows. First, the procedure checks to see if the two adjacent chopsticks are available to be picked up. If either is already picked up, the PickUp procedure fails. If both are available, the list of philosophers waiting for chopsticks is inspected to see if either neighbor has
waited longer. If neither neighbor has waited longer, the PickUp procedure succeeds.

When the PickUp procedure succeeds, two further steps are taken. First, the philosopher removes himself from the waiting list. Next, the array of boolean variables is updated to indicate that the two chopsticks are taken.

When the PickUp procedure fails, the philosopher checks to see if he is located on the waiting list. If he is not, he adds himself to the list.

The activity of each philosopher is represented by a process type. Two parameters are passed to this process type: a philosopher identification number, used to indicate the philosopher’s position at the table; and the chopstick handling resource.

The philosopher process executes as an infinite loop. First, the philosopher thinks. When he becomes hungry, he repeatedly tries to pick up chopsticks until he is successful. Next, he eats. Finally, he returns the chopsticks to the table, enabling his neighbors to eat. This sequence is repeated infinitely.

The main program begins by initializing the chopstick handling resource. A parallel statement is then executed, invoking the philosopher processes.
program DiningPhilosophers;

const

    {*** NumPhilosophers is the number of philosophers. There are also NumPhilosophers chopsticks. ***}
    NumPhilosophers = 10;

type

    {*** A Resource type for handing out chopsticks. ***}
    ChopstickHolder = resource
        interface
            procedure Initialize;
            procedure Pickup(PNum : integer; ret Success : boolean);
            procedure PutDown(PNum : integer);
        implementation
            var

                {*** An array of type boolean is used to keep track of each chopstick. If an element is true, the corresponding chopstick is available. ***}
                ChopSticks : array [1..NumPhilosophers] of boolean;

                {*** An array of type integer is used to keep track of each philosopher that tried unsuccessfully to eat. This array is an ordered list, with the first element representing the philosopher that has waited the longest. WaitCount represents the number of unsuccessful philosophers. ***}
                WaitingPhilosophers : array [1 .. NumPhilosophers] of integer;
                WaitCount : integer;

procedure Initialize;
var
    i : integer;
begin
    {*** Make all the chopsticks available. ***}
    for i := 1 to NumPhilosophers do
        ChopSticks[i] := True;
    {*** Initially, no one is waiting. ***}
    WaitCount := 0;
end;

procedure AddToWaitList(Num : integer);
var
i : integer;
Found : boolean;

begin

{*** This procedure checks the waiting list to see if Num is
on the list. If it is not, it is added to the list. ***}
Found := False;
for i := 1 to WaitCount do
  if WaitingPhilosophers[i] = Num then Found := True;
if not Found then begin
  WaitCount := WaitCount + 1;
  WaitingPhilosophers[WaitCount] := Num;
end;
end;

procedure RemoveFromWaitList(Num : integer);
var
  i : integer;
  Found : integer;
begin
{*** This procedure checks the waiting list to see if Num is
on the list. If it is, it is remove from the list. ***}
Found := 0;
for i := 1 to WaitCount do
  if WaitingPhilosophers[i] = Num then Found := i;
if Found <> 0 then begin
  for i := Found to WaitCount - 1 do
    WaitingPhilosophers[i] := WaitingPhilosophers[i + 1];
  WaitCount := WaitCount - 1;
end;
end;

function WaitPosition(Num : integer) : integer;
var
  i : integer;
begin
  WaitPosition := NumPhilosophers + 1;
  for i := 1 to WaitCount do
    if WaitingPhilosophers[i] = Num then WaitPosition := i;
end;

function OthersWaitingLonger(Num : integer) : boolean;
var
    Right,
    Left,
    MyPosition : integer;

begin
    Left := Num - 1;
    if Left = 0 then Left := NumPhilosophers;
    Right := Num + 1;
    if Right > NumPhilosophers then Right := 1;
    MyPosition := WaitPosition(Num);
    OthersWaitingLonger := (WaitPosition(Right) < MyPosition)
                          OR (WaitPosition(Left) < MyPosition);
end;

procedure Pickup{(PNum: integer; ret Success : boolean)};
var
    Left : integer;

begin
    {*** Check if both chopsticks are available AND neither
        neighbor has been waiting longer. ***}
    Left := PNum - 1;
    if Left = 0 then Left := NumPhilosophers;
    if ChopSticks[PNum] AND ChopSticks[Left] AND Not
        OthersWaitingLonger(PNum)
    then begin
        {*** SUCCESS! Remove yourself from the waiting list
            and take the chopsticks. ***}
        RemoveFromWaitList(PNum);
        Success := True;
        ChopSticks[PNum] := False;
        ChopSticks[Left] := False;
    end else begin
        {*** FAILURE! Add yourself to the waiting list. ***}
        AddToWaitList(PNum);
        Success := False;
    end;
end;

procedure PutDown{(PNum: integer)};
var
    Left : integer;

begin
    {*** Return the appropriate chopsticks. ***}
    Left := PNum - 1;

if Left = 0 then Left := NumPhilosophers;
ChopSticks[PNum] := True;
ChopSticks[Left] := True;
end;
end; {*** End of resource declaration. ***}

Philosopher = process(PhilNum : integer; StickHolder : ChopstickHolder);
var
  GotSticks : boolean;
begin
  while TRUE do begin {*** Infinite Loop ***}
    {*** Thinking statements ***}
    {*** Done thinking. Now, get chopsticks. ***}
    repeat
      StickHolder.PickUp(PhilNum, GotSticks);
      until GotSticks;
    {*** Eating statements ***}
    {*** Done eating. Now, put down chopsticks. ***}
    StickHolder.PutDown(PhilNum);
  end; {*** End of infinite loop ***}
end;
end; {*** End of process definition ***}

var {*** Main Program variables. ***}
  StickHolder : ChopstickHolder;
  Philosophers : array [1..NumPhilosophers] of Philosopher;
  i : integer;
begin {*** Main program ***}
  {*** Put down all the sticks. ***}
  StickHolder.Initialize;
  {*** Start all the philosopher processes. ***}
  parallel
    for i := 1 to NumPhilosophers do
      Philosophers[i](i, StickHolder);
  end;
end.
Chapter IV

Compiler Description

This chapter will discuss the operation of the compiler for the BPL language. Where appropriate, data structures and simplified Pascal coded algorithms are presented.

4.1 Compiler Overview

This compiler may be considered as three interdependent program modules, each of which concurrently perform a specified task during the compilation of a BPL program. These modules are the lexical analyzer, the syntactic analyzer, and the code generator. Another important entity of the compiler is the data storage table system maintained by the syntactic analyzer.

During the compilation of a program, the syntactic analyzer controls the overall operation of the compiler. The syntactic analyzer repeatedly calls upon the lexical analyzer to produce the next input symbol found in the source code file. Based upon the symbols found, the syntactic analyzer stores values in the data storage tables and calls upon the code generator to produce executable code. The syntactic analyzer contains procedures for parsing each BPL language construct.
The lexical analyzer translates the user source code into a sequence of tokens. The syntactic analyzer can more easily work with tokens than with the ASCII character strings used to represent BPL reserved words, expression operators, punctuation, and user defined identifiers. The syntactic analyzer maintains a hash table to efficiently manipulate the large variety of input symbols encountered in a BPL program.

The code generator responds to procedural calls from the syntactic analyzer. Each procedure is equivalent to an intermediate language instruction. The syntactic analyzer provides parameters with each intermediate language code request, enabling the code generator to generate code corresponding to the appropriate BPL data types and programming structures. The code generator utilizes a virtual operand stack. One entry is created on the stack for each element of an expression. Code is generated as virtual operands are removed from this stack.

The complete source code for the BPL compiler is located in Appendix A.

4.2 Lexical Analysis

The lexical analyzer performs the task of reading the program source code and translating the characters of the original program into tokens which can be easily processed by other portions of the compiler. The efficiency of the lexical analysis module is critical to the overall speed of the compiler due to the fact that long streams of characters must be digested and analyzed in order to correctly
determine the symbols, constants, reserved words, and identifiers which form a program.

The lexical analyzer is called by the syntactic analyzer through a procedure named Get. This procedure modifies a variable named LexToken, which represents the next symbol, identifier, or reserved word in the source program. The form of the LexToken's data type is shown below:

```pascal
typedef
TokenSymType = (Begin1, End1, If1, Then1, Else1, While1, Do1,
Repeat1, Until1, For1, To1, Downto1, Const1, Var1,
Type1, Procedure1, Function1, Plus1, Minus1, Star1,
Divide1, Modulo1, Equals1, NotEquals1, LessThan1,
GreaterThan1, LessEqual1, GreaterEqual1, Period1,
Dotdot1, Comma1, Colon1, Semicolon1, LeftParen1,
RightParen1, LeftBrkt1, RightBrkt1, Identifier1,
IntConstant1, CharConstant1, StringConstant1,
RealConstant1);

TokenType = record
  Sym : TokenSymType;
  IntVal : integer;
end;

var
  LexToken : TokenType;
```

TokenSymType is an enumerated type with a different element for each possible element in a source program. The above is an abbreviated type definition; the actual definition of TokenSymType contains over one hundred enumerated elements. TokenType is a record type with two fields. The first field, Sym, is of type TokenSymType and describes what type of token was encountered in the
source program code. For many types of tokens, such as the reserved word tokens begin1 and end1, this is sufficient to completely describe the lexical component of the program. For other types of tokens, such as user specified identifiers or integer constants, the field IntVal is used to further identify specific properties of the token.

A user specified identifier token, for example, contains an integer value within the IntVal field which uniquely identifies the identifier. All references to a given identifier will result in an identifier token with the same value assigned to the IntVal field. An integer constant token contains the numeric value of the constant within the IntVal field. String constants and real constants are stored in tables which are maintained by the compiler; for these lexical components, the IntVal field is a index into the string table or the real table, respectively.

The lexical analyzer maintains a list of all identifiers and reserved words used in a program. It is critical to the overall efficiency of the lexical analyzer that this list can be searched quickly. Additionally, the storage method for identifiers and reserved words must be efficient in terms of memory usage since several thousand identifiers could potentially exist in a given program.

One method considered for storing identifiers and reserved words is an array of strings, where each identifier or reserved word is represented by one element of the array. Such an array permits reasonably fast searches for a particular input string. This approach, however, presents several drawbacks. For example, when declaring an array of strings, an arbitrary maximum string length
must be enforced. Therefore, some upper limit must be specified for the number of significant characters in an identifier or reserved word. This is undesirable, since longer variable names can improve the readability of a program. Furthermore, by declaring an array of strings, each identifier or reserved word would require the same amount of storage space regardless of the number of characters in each string. Therefore, much memory would be wasted since many identifiers consist of very few characters.

In order to satisfy the two criteria of efficient memory usage and quick table searching, two separate data structures are used to keep track of the identifiers and reserved words. The first of these is CharTableType:

```plaintext
type
  CharTableType = array [1..MaxNumChars] of char;
```

A variable of this type is used to store all ascii characters of each reserved word and each user identifier used in a program. Initially, this variable is empty. As new identifiers are encountered, they are added to this array of characters. This storage method is very efficient, since each character of an identifier requires only one byte of memory space.

For example, assume three user identifiers, of length one, two, and four characters, respectively, are encountered within a program. The first identifier's one character name would be stored in the first element of the character array. The second identifier's two character name would be stored in the second and third elements of the character array. The third identifier's four character name
would be stored in the fourth through seventh elements of the character array.

Obviously, this storage technique is not sufficient to completely describe an identifier or reserved word. Further information must also be stored to indicate 1) where the first character of a particular string lies within the character array and 2) how many characters are in the string. This information is stored in IdentifierTable, an array of type SymbolDescType:

```plaintext
type
  SymbolDescType = record
    CharTablePtr : longint; {Char table index}
    WordLength : longint; {String Length}
    HashVal : longint; {Hash value for string}
    SymbolType : LexSymbol; {Describes type of string}
    Valid : boolean; {Is entry in use?}
  end;

var
  IdentifierTable : array[0..MaxNumIdentifiers] of SymbolDescType;
```

The first field in this record type, CharTablePtr, indicates where the first character of the identifier or reserved word is within the character table. The second field, WordLength, represents the number of characters in the identifier or reserved word.

The third field, HashVal, contains an integer hash function value which is computed as a weighted sum of the ascii values of the characters in the string. The maximum value of this hash function is the value MaxNumIdentifiers. This value is used when searching for an identifier within the table, as will be discussed in a following section.
The next field, SymbolType, is used to indicate what type of token the input string represents. Prior to opening and reading the user's source code program file, the compiler inserts all reserved words into the identifier table. For example, the reserved words "begin" and "end" are inserted into the identifier table. The SymbolType field for each of these reserved words would have the values begin1 and end1, respectively. Upon reading the source program file, the compiler assumes that any unknown token encountered in the text is a user-declared identifier. Therefore, the SymbolType field for any previously undefined token encountered in the program contains the value identifier1.

The last field, Valid, is a boolean field used to represent whether or not a reserved word or user-declared identifier is stored in this element of the identifier table. This is necessary because identifiers are not stored in the array variable IdentifierTable sequentially. Instead, they are distributed throughout the array depending upon the hash function value for the string to be stored.

Below is the abbreviated Pascal code for the procedure that inserts a reserved word or identifier into the variable IdentifierTable:
procedure InsertIdent(Id:string;IdentType:LexSymbol);
var
    HashFuncVal : longint;
    IdTablePtr : word;
    I : word;
begin
    { Convert the Identifier to Upper Case.}
    Id := UpperCase(Id);

    {Calculate the Hash Function for the Identifier.}
    HashFuncVal := HashFunction(Id);

    { Initialize the IdTablePtr to the HashFnct Value.}
    { This is the first position we will try to store in}
    IdTablePtr := HashFuncVal;

    { Search for a location to store in the ID Table. }
    while IdentifierTable[IdTablePtr].Valid = true do
        IdTablePtr := (IdTablePtr + 1) mod MaxNumIdentifiers + 1;

    { At this point, we have found a suitable location.}
    { Now, store the appropriate data. }
    with IdentifierTable[IdTablePtr] do begin
        { Indicate this location is now in use. }
        Valid := true;

        { Store the Starting position & length of chars.}
        CharTablePtr := NextCharPosition;
        WordLength := length(Id);

        { Store the Hash Function Value for the ID. }
        HashVal := HashFuncVal;

        { Indicate the type of identifier this is. }
        SymbolType := IdentType;
    end;

    { Finally, store the characters of the ID into }
    { the character table. }
    for i := 1 to length(Id) do begin
        CharTable[NextCharPosition] := Id[i];
        NextCharPosition := NextCharPosition + 1;
    end;
end;

The technique used by this procedure is fairly straightforward. First, the string is converted to all upper case characters. This makes the compiler case-insensitive. The hash function then is calculated for the string. This is performed by calling a function which calculates a simple weighted sum of the ascii codes of the characters. The result of this calculation is in the range 0..MaxNumIdentifiers. Next, the procedure finds an "empty" location in the identifier table to store the data for this string. A location is identified as empty if the Valid field of that element is false. The first location tried is the element of the array pointed to by the hash function value. If that location is empty, the data for the string is stored there. Otherwise, the next location is tried. This is repeated until an empty location in the identifier table is located. All data for the string is then stored into this location, including the hash function value, the location in the character table of the input string's first character, the number of characters in the input string, and the type of token the input string represents. Additionally, the Valid field of this location is set to true, indicating that the element is no longer empty. Finally, each character of the input string is copied into the character table.

Before processing source code, the compiler must initialize the identifier table and the character table. The following procedure is used to perform this task:
procedure InitIdData;
var i : word;
begin
  { Set the Character Table Pointer to indicate that }
  { no characters have been stored. }
  NextCharPosition := 1;

  { Set all Valid fields in the identifier table to }
  { false, indicating that all elements are empty. }
  for i := 0 to MaxNumIdentifiers do
    IdentifierTable[i].Valid := false;

  { Finally, insert all reserved words into the }
  { identifier table. }
  InsertIdent('begin',begin1);
  InsertIdent('end',end1);
  InsertIdent('if',if1);
  InsertIdent('then',then1);
  InsertIdent('else',else1);
  InsertIdent('while',while1);
  InsertIdent('do',do1);
  InsertIdent('repeat',repeat1);
  InsertIdent('until',until1);
  { ... and so on for all reserved words. }
end;

At the completion of this procedure, all reserved words are represented within the identifier table. Another required procedure is one which finds an entry in the identifier table matching a string found in the input file. If this procedure is unsuccessful in finding the input string, the string is added to the identifier table as a user-declared identifier.
procedure FindIdent(Id:string;var Token : TokenType);
var HashFuncVal : word;
   IdTablePtr : word;
   Found : boolean;
   TmpStr : string;
   i : word;
begin
   {Convert the Id string to upper case.}
   Id := UpperCase(Id);

   {Calculate the hash function value of the Id.}
   HashFuncVal := HashFunction(Id);

   {This is where we start looking.}
   IdTablePtr := HashFuncVal;
   Found := false;

   { Keeping looking until a match is found or we }
   { determine that there is no match. }
   while not Found and
   (IdentifierTable[IdTablePtr.Valid = True) do
      with IdentifierTable[IdTablePtr] do begin
         { See if the hash function of the ID }
         { matches the hash function of the location }
         { we are currently checking for a match. }
         if HashVal = HashFuncVal then begin
            { The hash functions compare. Next, }
            { extract the string from the character }
            { table and check for a match. }
            TmpStr := ";
            for i := CharTablePtr to CharTablePtr +
               WordLength - 1 do
               TmpStr := TmpStr + CharTable[i];
            if TmpStr = Id then Found := true;
         end;
         { If no match, then increment the IdTablePtr to}
         { the next storage location. }
         if not Found then IdTablePtr := (IdTablePtr + 1) mod
         (MaxNumIdentifiers + 1);
      end;
   { If we get here and Found is true, we have located}
   { the identifier.}
   if Found then begin
      { Indicate the type of the identifier and the }
{ IntVal associated with it. }
Token.Sym := IdentifierTable[IdTablePtr].SymbolType;
Token.InvVal := IdTablePtr;
end else begin
{ If we get here, we have not been successful. }
{ Store the Id as a user-declared identifier. }
InsertIdent(Id,identifier1);

{ Now, we must store the appropriate values in }
{ Token. We know that the Sym Type is identifier }
Token.Sym := identifier1;

{ We need not search again for the storage position}
{ in the identifier table. The search failed at }
{ the element pointed to by IdTablePtr. The }
{ InsertIdent procedure will therefore insert the}
{ Identifier data at the same position . . . }
Token.InvVal := IdTablePtr;
end;
end;

This procedure searches the identifier table for a match to the current input string. It starts by calculating the hash function value for the input string. Then, it begins searching the identifier table at the element pointed to by the hash function value. If the identifier stored at that position matches the input string being searched for, the variable Found is set to true. Otherwise, the search pointer is incremented to point to the next element in the identifier table. This continues until either a match is found or an empty position is encountered in the identifier table. If an empty position is encountered, the input string is not currently stored in the identifier table. It therefore must be added. This is performed by calling the procedure InsertIdent. Note that, if the input string is not found, the compiler assumes that it is a user-declared identifier and stores it as such.
Finally, after either finding or inserting the input string, information regarding the input string is returned to the caller. This includes the type of the input string and an integer value used to further identify the string. For reserved words, such as "begin" and "end", the integer value is not necessary. The integer value is necessary, however, to distinguish between various user-declared identifiers. The integer value returned for each search is the location in the identifier table where the input string is stored. This number is suitable for this use since it will be unique for each input string.

The following variable and procedure are used extensively by the lexical analyzer:

```
var
  CurrentChar : char;

procedure GetNextChar;
begin
  read(InputFile, CurrentChar);
end;
```

CurrentChar is a variable which always contains the current input character in the input stream. GetNextChar is a procedure used to read the next character from the input stream into CurrentChar, effectively discarding the previous value of CurrentChar.

The main procedure of the lexical analyzer is called Get. Get scans the character stream from the input file and determines what the next input token. The result is stored in the variable LexToken, which is of type TokenType. The
Get procedure does little processing; instead it examines the next input character, stored in CurrentChar, and calls an appropriate routine depending upon what the character is. For example, if CurrentChar is a decimal digit, Get calls a routine used to read in a numeric value. If CurrentChar is a single quotation, Get calls a routine used to read in a string or character constant. Each of these routines will read in the required number of characters, determine the necessary data for the source code token, and store the result in the variable LexToken. The Pascal code for Get is as follows:
procedure Get;
begin
{ Skip over any spaces, tabs, CR's, etc. }
while CurrentChar in [' ',TAB,CR,LF] do
  GetNextChar;

{ Call a procedure depending upon CurrentChar. }
case CurrentChar of
  '0'..'9' : ScanIntegerConstant;
  'A'..'Z',
  'a'..'z' : ScanRsrvdWordOrId;
  ',', ',', '<', '>',
  '/', ',', '=' ,'+',
  '*', '(' ,')', ',','
  ':', '[' ,']'    : ScanSpecialSymbol;
  SingleQuote : ScanStringOrCharConst;
  '{' : begin
    ScanComment;
    Get;
  end;
else FlagError;
end;
end;

Much of the simplicity of this procedure is due to the specification of Pascal. Any input token beginning with a letter must be either a reserved word or a user-declared identifier. A numeric digit at the beginning of a token indicates a numeric constant. The other types of tokens follow similar rules. The first statement in the Get procedure skips over any spaces, tabs, carriage returns, or line feed; this is in accordance with the Pascal specification, which states that white space can be used freely without any effect on the program's behavior.
Each of the procedures called by Get must strictly adhere to some simple rules. The procedure must read only the new characters that are necessary for the token being scanned. After reading in the token, the variable LexToken must be modified to indicate the type of the token. Finally, the variable CurrentChar must contain the input character immediately following the processed token in the input file.

The ScanComment procedure does not strictly adhere to these rules because ScanComment does not modify the variable LexToken. The reason for this is that a comment within the user's source code does not constitute an input token. Therefore, after the procedure Get calls the procedure LexToken, Get recursively calls itself again in order to obtain the next input token of significance and store it in the variable LexToken.

Each of the token scanning procedures will now be discussed. The procedure ScanIntegerConstant is used to read in the characters from the input file which comprise an integer constant. The value of the constant is calculated. The variable LexToken is modified so that the Sym field has a value of IntConstant1, and the IntVal field is assigned the value of the integer constant.
procedure ScanIntegerConstant;
var Value : longint;
begin
  { Initialize Value to the numeric value of the }
  { first digit of the number. }
  Value := ord(CurrentChar) - ord('0');

  { Read in the next character. }
  GetNextChar;

  { Accumulate a sum in Value as long as the input }
  { character is a decimal digit. }
  while CurrentChar in ['0'..'9'] do begin
    Value := 10 * Value + ord(CurrentChar) - ord('0');
    { Read in the next character. }
    GetNextChar;
  end;

  { At this point, CurrentChar is not a dec. digit. }
  { CurrentChar therefore holds the first character }
  { of the NEXT token in the input file. }

  { Store the results in LexToken. }
  LexToken.Sym := IntConstant1;
  LexToken.IntVal := Value;
end;

The ScanRsvdWordOrId procedure concatenates a string consisting of a stream of input characters. The string is accumulated as long as the current input character is a letter, a decimal digit, or the underline character. Then, the FindIdent procedure is called to find the string in the identifier table. If the string is not found, FindIdent will insert the string into the table. The global variable LexToken is passed as a parameter to FindIdent; FindIdent will store the appropriate values here indicating what type of reserved word or user-declared identifier the input string is. The variable CurrentChar contains the first character
encountered in the input stream which is not part of the reserved word or identifier. This is, therefore, the first character in the next input token.

```pascal
procedure ScanRsrvdWordOrId;
var IdString : string;
begin
  { Initialize the IdString. }
  IdString := CurrentChar;

  { Get the next character. }
  GetNextChar;

  { Accumulate the string. }
  while CurrentChar in ['A'..'Z','a'..'z','0'..'9','_']
    do begin
      IdString := IdString + CurrentChar;
      GetNextChar;
    end;

  { Find or ins. the string, store result in LexToken.}
  FindIdent(IdString,LexToken);
end;
```

The ScanSpecialSymbol procedure is used to scan the input stream for the variety of punctuation marks and operators that are used in the Pascal language. Many of these symbols are trivial to process, such as the plus symbol ('+'). For the plus symbol, LexToken.Sym is assigned a value of plus1 and GetNextChar is called so that CurrentChar contains the first character of the next token. For other symbols, however, the process is not quite so simple. Consider the less than character ('<'). This character can appear alone to represent less than. It can also be the first character of a less than or equal symbol ('<=') or a not equal symbol ('<>'). For these types of symbols, the next character must be examined. If the
next character is part of a two character symbol, that type of symbol is indicated. Otherwise, the single digit version is indicated. In all cases, upon termination of the ScanSpecialSymbol procedure CurrentChar must contain the first character of the next token in the input stream.
procedure ScanSpecialSymbol;
var Char1 : char;
begin
  { Save the current character, get the next one. }
  Char1 := CurrentChar;
  GetNextChar;

  { Now, process the input based upon Char1. }
  case Char1 of
    '+': LexToken.Sym := Plus1;
    '-': LexToken.Sym := Minus1;
    ':': LexToken.Sym := Semicolon1;
    '(' : LexToken.Sym := LeftParen1;
    ')': LexToken.Sym := RightParen1;
    '[' : LexToken.Sym := LeftBrkt1;
    ']': LexToken.Sym := RightBrkt1;
    ',': LexToken.Sym := Comma1;
    '*': LexToken.Sym := Star1;
    '/': LexToken.Sym := Divide1;
    '=': LexToken.Sym := Equals1;
    ':': if CurrentChar = '=' then begin
      LexToken.Sym := Assign1;
      GetNextChar;
    end else LexToken.Sym := Colon1;
    '>': if CurrentChar = '=' then begin
      LexToken.Sym := GreaterEqual1;
      GetNextChar;
    end else LexToken.Sym := GreaterThan1;
    '.': if CurrentChar = '.' then begin
      LexToken.Sym := DotDot1;
      GetNextChar;
    end else LexToken.Sym := Period1;
    '<': if CurrentChar = '=' then begin
      LexToken.Sym := LessEqual1;
      GetNextChar;
    end else begin
      if CurrentChar = '>' then begin
        LexToken.Sym := NotEqual1;
        GetNextChar
      end else LexToken.Sym := LessThan1;
    end;
  end;
end;
The procedure `ScanStringOrCharConst` is used to assemble string and character constants from the source code. The procedure simply builds a string of all characters in the input file up to but not including the single quote delimiter. Next, the string is investigated to determine if it is actually a character constant. If it is a character constant, `LexToken.Sym` is set to `CharConst1`, while `LexToken.IntVal` is set to the ascii value of the character constant. If it is not a character constant, a special procedure is called which stores the string constant in a string table. This procedure returns the index number of the string table element in which the string constant was stored. This value is stored in `LexToken.IntVal`.

```pascal
procedure ScanStringOrCharConst;
var Str : string;
begin
  { Initialize Str, skip past the initial single quote.}
  Str := ";
  GetNextChar;

  { Build up the string, until a single quote is encountered. }
  while CurrentChar <> SingleQuote do begin
    Str := Str + CurrentChar;
    GetNextChar;
  end;

  { See if Str is of length 1 => char const }
  if length(Str) = 1 then begin
    LexToken.Sym := CharConst1;
    LexToken.IntVal := ord(Str[1]);
  end else begin
    LexToken.Sym := StringConst1;
    StoreStringInTable(Str,LexToken.IntVal);
  end;
end;
```
The ScanComment procedure is used to read and ignore all characters associated with program comments. This procedure allows for nested comments. The basic algorithm involves counting the nesting level of the comments. Whenever a beginning of a comment level is detected, the counter is incremented. Whenever an end of comment level is detected, the counter is decremented. This continues until the counter is zero.

```pascal
procedure ScanComment;
var NestLevel : word;
begin
  { Initialize NestLevel. }
  NestLevel := 0;

  { Read in characters until the comment is over. }
  { Note that the first time through this loop, }
  { CurrentChar = '{'. Therefore, NestLevel will }
  { be set to one (1) during the first iteration. }
  repeat
    if CurrentChar = '{' then inc(NestLevel);
    if CurrentChar = '}' then dec(NestLevel);
    GetNextChar;
  until NestLevel = 0;
  { At this point, the comment has been skipped over }
  { and CurrentChar now contains the first character }
  { of the next token. }
end;
```

In the procedure Get, if CurrentChar is none of the choices listed in the case statement, the procedure FlagError is called using a parameter indicating the occurrence of an illegal character in the source code text. This will cause the compiler to halt and give an error message indicating that an illegal character was encountered in the source code file.
The internal operation of the lexical analyzer is transparent to all other components of the compiler. The only interface to the lexical analyzer is the procedure Get and the variable LexToken. The syntactic analyzer will repeatedly call this procedure and examine the contents of this variable during the compilation of a program.

4.3 Compiler Tables

Several tables are maintained by the compiler in order to maintain information regarding all program components. The structure of each of these tables will be discussed in this section, with a description of the data types used to represent the tables and a discussion of the usefulness of the values stored in the tables. Following the description of each table’s structure, the procedures used to modify and access these tables will be discussed.

4.3.1 Compiler Table Structure

The first table to be discussed is the identifier table. The identifier table is referenced by using the variable IdTabl, which is an array of type IdRec. Each element in the array describes a single identifier. A variable, IdTablPtr, contains the number of identifiers stored in IdTabl. The type IdRec references another type, IdDesc. These types are presented below:
type
   IdDesc = (LongintConst, CharConst,
            StrConst, BoolConst, EnumConst,
            RealConst, TypeName, ProcName,
            FuncName, VarName, ValParamName,
            VarParamName, RetParamName,
            FwdProcName, FwdFuncName);

   IdRec = record
             IdTyp : IdDesc; \{type of id\}
             IdNum : longint; \{name of the identifier\}
             level : shortint; \{0=main block, etc\}
             valptr : longint; \{misc value/pointer\}
         end;

The first element of the IdRec type is IdType. This field is used to indicate what the identifier represents within the BPL program. The possible types of identifiers include the following:

1) Constant names, declared with a constant declaration. Constants can be of type long integer, character, string, boolean, a user-specified enumerated type, or real.

2) Type names. These are created within a type declaration, where a user-specified type becomes associated with an identifier.

3) Procedure, function, forward procedure, and forward function names. These are added to the identifier table as user-declared procedures and functions are encountered. BPL allows the use of forward-defined procedures and functions. These are entered into the identifier table as forward procedures and forward functions. When the actual procedure or function is declared, the IdType field is modified to make it a standard procedure or function.

4) Variable, val parameter, var parameter, and ret parameter names. These names are added to the identifier table when they are encountered within variable declarations, for variable names, or within procedure, function, or process declarations for parameter names.
The IdNum field is used to store the integer used to represent the identifier. This integer value is assigned by the lexical analyzer. With the syntactic analyzer, all identifiers are represented only by this value. The level field is used to indicate the block level in which an identifier is declared. Identifiers declared within the main program block have a level value of zero (0). Identifiers declared within a procedure declared within the main program block have a level value of one (1). Identifiers declared within procedures nested within other procedures have level values of two (2) or greater.

The block level of an identifier is useful for several reasons related to the scope of Pascal programs. All identifiers declared at a given level must be unique. Therefore, before declaring an identifier, IdTabl must be searched to see if the same identifier has been declared at the same level. This is accomplished by search the table for an entry with the same identifier value and the same block level value.

Pascal allows identifiers to be redeclared at different block levels. For example, one identifier can be used to specify a variable in the main program and a constant within a procedure. Identifiers are added to the identifier table at the end. To find a given identifier, the table is searched backwards starting at the end. The first occurrence of the identifier will correspond to the highest block level in which the identifier is declared.

Identifiers declared within a nested program entity are invisible outside of the entity. In order to accommodate this, all identifiers declared within a nested
entity are removed from the identifier table at the completion of the entity. This is accomplished by decrementing IdTablPtr until all identifiers associated with the terminating block level are removed. Because of the block structure of Pascal, all such identifiers will be at the end of the table.

The final field in the IdDesc type is valptr. As the name suggests, this field contains either a value or a pointer. For integer constants, character constants, enumerated constants, and boolean constants, valptr contains the ordinal value of the constant. For real and string constants, valptr contains a pointer into an array of real and string constants, respectively. For type names, valptr points to an entry in the type table. For procedure names, valptr points to an element of the procedure table. For function names, valptr is the index of an element in the function table. For variables and parameters, valptr contains the index of an element in the variable and parameter tables, respectively. Each of these tables contains all necessary information about types, procedures, functions, variables, and parameters, respectively. These tables will be described below.

The type table is used to describe all standard and user-declared types encountered in the user program. This table is stored in the variable TypeTabl, which is an array of type TypeDataType. Each element in the array describes a single data type. A variable, TypeTablPtr, contains the number of type descriptions stored in TypeTabl. The type TypeDataType references another type, TypeDescType. These types are presented below:
type
  TypeDescType = (IntType, LongintType, ShortintType, ByteType,
  WordType, RealType, DoubleType, SingleType, PtrType,
  FwdPtrType, CharType, BoolType, StringType, ArrayType,
  RecordType, SubrangeType, EnumType, FileType, TextType,
  ChannelType, ProcessType, ResourceType, TemplateType,
  SharedType);

TypeDataType = record
  TypeDesc : TypeDescType;
  SubType : longint;
  p1,p2 : longint; {misc. pointers and/or values}
  VariableSpace : longint; {var size for resource}
  TypeSize : longint; {size of the data type}
  AddrMask : longint; {mask used for alignment}
  level : shortint; {current block level}
end;

The first field in TypeDataType is used to indicate the class of data type
being described. These are listed above in the TypeDescType type declaration. All
are self-explanatory, with the exception of the FwdPtrType type specification. In
Pascal, types can be declared which are pointers to undefined types. These
undefined types are later declared. During the interval between the forward
pointer type reference and the actual declaration of the type being pointed to, the
pointer type is declared with a type FwdPtrType. This is changed when the
undefined type is declared.

The field TypeSize is used to store the number of bytes required to store a
variable. For the standard types, this is a fixed number, as follows:
longint, pointers, real : 4 bytes
integer, word : 2 bytes
byte, shortint, boolean, char, enum : 1 byte
double : 8 bytes
strings : 256 bytes
file, text : 90 bytes

For other types, the size is dependent upon the type. For example, an array type size is dependent upon the size of the element and the number of elements for the type.

The AddrMask field is used to properly align variables stored in memory. The value of this field is a bit mask which indicates which bits in the memory address for a variable of this type must be zero. For example, a long word in a 680x0 microprocessor's memory system will be written or read more efficiently if the address is long word aligned. This occurs when bits zero and one of the memory address are zero. Therefore, the value of AddrMask for a longint type is three (3). Byte values, on the other hand, are stored with equal efficiency at any address. The value of AddrMask for a byte type is therefore zero (0). Word length data types have an AddrMask value of two (2).

The AddrMask for an array type is equal to the AddrMask of the array element. Therefore, an array of longint elements would have an AddrMask of three (3), while an array of bytes would have an AddrMask of zero (0). The AddrMask for a record type is calculated by logically OR'ing the AddrMask value for each field type contained within the record. Consider the following type declaration:
SampleRec = record
  x : byte;
  y : longint;
end;

The AddrMask values for the two fields have values of zero and three, respectively. Logically OR'ing these values gives an overall AddrMask for the record type of three.

In record types, the offset of each field is adjusted according to the value of AddrMask for each field's type. In the above example, without AddrMask correction, x would have an offset of zero, and y would have an offset of one. This situation would result in y not being long word aligned. Therefore, each offset is adjusted using the following formula:

\[
\text{AdjustedOffset} = (\text{offset} + \text{AddrMask}) \text{ AND (not AddrMask)}
\]

Using this formula, the adjusted offset for field y in the above example would be four. The overall size of the above type would be eight bytes.

In array types, the spacing between elements must also be adjusted using the above formula. Consider the following type:

\[
\text{SampleArray} = \text{array [1..5] of record}
  y : longint;
  x : byte;
end;
\]

The element size for this array without address offset compensation is five bytes. If the elements were spaced by five bytes, however, not all elements would be long word aligned. If the above array began at address zero, the second array element
would have an address of five, the third ten, and so on. By using the above formula, the proper spacing is eight bytes.

When variables are declared, they are stored consecutively in memory in the order in which they appear. The first variable is assigned an address which is the base address for variable storage. The second address is stored after the first. The address of the second variable is calculated by adding the address of the first variable to the size of the first variable, and adjusting the result with the above equation using the AddrMask value of the type of the second variable.

It should be noted that on the 680x0 microprocessor, properly aligning operands is not necessary, but does improve program efficiency. On other processors, such as the Inmos Transputer, operands must be properly aligned within memory; otherwise, a run-time error will occur or inaccurate data will be read or written.

The field level is used to store the block level in which the type is declared. This field has no real meaning with regard to usage of the type. At the end of each nested program entity, however, types defined within the entity can be discarded, possibly conserving memory storage space. The level field is used to perform this task. Additionally, at the end of each nested entity, any forward pointer type declared within the entity must have been resolved. This is checked while discarding nested type definitions, and the error is reported if any are detected.

The remaining fields of the TypeDataType are dependent upon the value of
TypeDesc. These will now be described for each possible value of TypeDesc.

The simple types are longint, integer, word, shortint, byte, real, double, char, string, and boolean. For these types, no other values are stored in the remaining fields. A subrange type is similar to a longint, integer, word, shortint, or byte type, depending upon the maximum and minimum values. The smallest of these simple types is selected which can represent all possible value within the specified range. The field SubType is the index into TypeTabl of this selected type. The fields p1 and p2 are used to represent the minimum and maximum values of the specified range, respectively.

For pointer types, the field SubType is the index into TypeTabl of the type pointed to. For example, a pointer to an integer would have SubType equal to the index of the element in the type table containing the simple type integer.

For array types, the fields p1 and p2 are used to point to two other type definitions within the type table. P1 points to the index type of the array type, while p2 points to the element type of the array type. Consider the following examples:

```
array1 = array [1..5] of byte;
array2 = array [char] of array1;
```

For the first type, array1, the field p1 would point to a subrange type description. Within this subrange type description, SubRange would point to the simple type byte, p1 would contain the value 1, and p2 would contain the value 5. The field
p2 for array1 type description would point to the description of the simple type byte. For the second type, array2, the field p1 would point to the description of the simple type char. The field p2 would point to the type description of array1.

For record types, the field p1 is the index within the field table of the first field found in the record. (The field table will be discussed below.) The field p2 contains the number of fields in the record type. Therefore, within the field table, entries for the record type would reside in elements p1 through p1+p2-1.

For file types, the field SubType contains the index of the element type of the file. For example, a file of longint type would have a value of SubType that points to the type description of the simple type longint.

For process types, the field SubType contains the label number for the assembly language code that executes the corresponding process. For example, if the label for a process is L25:, the value of SubType for that process type would be 25. The field p1 represents the index into the parameter table of the first process parameter. The field p2 represents the number of process parameters. Therefore, within the parameter table, the entries for the parameters for the process would reside at elements p1 through p1+p2-1.

For resource types, the field SubType represents the amount of stack space required for the resource, and the field VariableSpace represents the amount of space required to store permanent variables for the resource. Both of these memory sizes are in bytes. The field p1 is the index into the field table for the first interface procedure for the resource. The field p2 is the number of interface
procedures for the resource. Therefore, within the field table, data for each interface procedure is stored in elements p1 through p1+p2-1.

For channel, shared, and template types, the only additional information required is the index of the element type within the type table. This index is stored in the field SubType.

The field table, mentioned above with regard to record and resource types, is stored in the variable FieldTabl, which is an array of type FieldDescType. This type is defined as follows:

```plaintext
type
  FieldDescType = record
    FName,
    FTypePtr,
    FTypePcnt,
    AddrOffs : longint;
  end;
```

These fields have different meanings for representing record fields and resource procedures. For record fields, the field FName contains an integer which uniquely identifies the name of the field. This number is assigned by the lexical analyzer. The field FTypePtr is an index into the type table which points to the type of the field variable. The field AddrOffs stores the offset into the record data type of the field component. The actual address of a record field of a record variable is calculated by adding AddrOffs to the base address of the variable.

Consider the following record type declaration:
type
  SampleRec = record
    x : longint
    y : char;
    z : byte;
  end;

Each field in this type would contain an entry in the field table. For the first field, x, the field Fname would contain the integer assigned to the name "x" by the lexical analyzer. The field Ftypeptr would contain an index into the type table which points to the longint type definition. The field AddrOffs would contain the value zero, since x is the first field in the variable. For the second field, y, the structure is similar. This time, however, the field Ftypeptr points to the definition of the char type. Also, type field AddrOffs would contain a value of four, which is the size of the first record field. Finally, the third field, z, would have a value of Ftypeptr that points to the definition of the type byte, and a value of five for AddrOffs. Note that if field z were of type longint that the AddrOffs would be long word aligned to a value of eight.

For representing resource procedures, the fields of the record type FieldDescType take on the following meanings:

The field Fname contains the integer assigned by the lexical analyzer to represent the name of the resource procedure. The field Ftypeptr contains the index into the parameter table of the first resource procedure parameter. (The parameter table will be discussed below.) The field Ftypepcnt contains the number of parameters declared for the resource procedure. Therefore, data describing the parameters for
the resource procedure are stored in elements Ftypeptr through (Ftypeptr + Ftypepcnt - 1) of the parameter table. Finally, the field AddrOffs contains the logical label number for the first assembly language instruction for the resource procedure. For example, if AddrOffs had a value of twenty-five, the assembly language code for the resource procedure would begin at label L25.

Data describing procedures and functions are stored in the procedure table and the function table, respectively. These tables are stored in variables ProcTabl and FuncTabl, which are respectively of type ProcEntryType and FuncEntryType:

```plaintext
type
  ProcEntryType = record
    level : byte;
    FirstParam,
    ParamCount,
    ParamSize,
    CodeLabel : longint;
  end;

  FuncEntryType = record
    level : byte;
    FirstParam,
    ParamCount,
    ParamSize,
    TypePointer,
    CodeLabel : longint;
  end;
```

These two types are identical, except that the type describing functions contains one extra field, TypePointer. This field is an index into the type table which points to the type definition for the result type of the function. This field is not needed to describe procedures since procedures do not return a result.
The field level is used to indicate the block level in which the procedure of function is declared. This is needed for several reasons. Firstly, as described above, the compiler discards all program entities declared within each block level at the end of that level. This saves space in the storage tables, and it enforces the rules of scope; external program components cannot access components declared within a nested block level if the nested components have been discarded.

The block level field is also used to determine if a procedure or function is declared internal or external to a process or resource. In BPL, all procedural calls inside processes and resources are limited to procedures declared within the process or resource. This condition can easily be determined by inspection of the block level field.

Another important need for the block level field involves the linking of nested procedures and functions. As will be discussed in the description of code generation, a stack frame is constructed for each procedure or function. In addition to storing procedural parameters and local variables, this stack frame also contains pointers to the variables and parameters of the surrounding block level. Consider as an example a nested procedure of block level three. If this procedure calls a nested procedure of block level four, the pointers stored into the stack frame for the called procedure must point to the parameters and local variables of the procedure at level three, the parent level of the called procedure. If, however, the level three procedure calls another level three procedure, the resulting stack frame must contain pointers to the parameters and variables of the parent
procedure at level two. If the level three procedure were to call a level two procedure, the stack frame must provide linkage to the parameters and variables of level one, and so on. In all cases, the calling procedure must know the block level of the function or procedure to be called in order to construct the appropriate stack frame. The field level within these types is used to provide that information.

The field FirstParam is the index into the parameter table of the first parameter for the procedure or function. The field ParamCount contains the number of parameters needed for calling the procedure or function. Therefore, data describing the parameters for the resource procedure are stored in elements FirstParam through (FirstParam + ParamCount - 1) of the parameter table.

The field ParamSize contains the size, in bytes, of the memory space required to store all the parameters for the procedure or function. For functions, however, this value also includes the space required to store the function result. Before calling a procedure or function, the calling program creates room on the stack for the parameters and, for a function, the function result. At the end of a procedure or function, the procedure or function removes the parameters from the stack. Functions, however, return leaving the function result on the stack. Therefore, the calling program must return the function result from the system stack.

Finally, the field CodeLabel contains the logical label number for the first assembly language instruction for the procedure or function. This label numbers works in the same manner as the label number for a resource procedure.
The variable table is used to describe all variables declared in the user program. This table is stored in the variable VarTabl, which is an array of type VarDataType. Each element in the array describes a single variable. A variable, VarTablPtr, contains the number of variable descriptions stored in VarTabl. The type VarDataType is defined as follows:

```plaintext
type
  VarDataType = record
    VarTypePtr, AddrOffset : longint;
    level : byte;
  end;
```

The field VarTypePtr contains the index of the element within the type table of the data describing the type of the variable. The field AddrOffset contains the offset, in bytes, of the variable within the variables of the appropriate block level. For example, all variables of block level zero are stored in a contiguous range of memory addresses, as are variables of block level one, two, and so on. For each block level, the beginning of this range of memory ranges can be determined either by inspecting a register within the CPU or by linking with parent block levels using the stack frame set up for each procedure or function. Once this starting address is determined, the address of the actual variable is calculated by adding the field AddrOffset to the base address. As with fields of a record type, AddrOffset is adjusted using the field AddrMask of the variable's type. Therefore, longint variables will always be long word aligned, word variables will always be word aligned, and so on.
Finally, the field level is used to indicate the block level of the variable. As with procedures and functions, this field serves several purposes. It is used to discard a local variables within a nested program level at the completion of that level. It is used to guarantee that a process or resource does not access variables external to the process or resource. Finally, the block level is used by the code generator in order to properly calculate the address of the variable. As mentioned above, all variables of a given block level are stored in a contiguous range of memory addresses. The address of a variable is determined by finding the base address of the variables for a given block level, then adding the address offset of the variable to this base address. The code generator must therefore be able to determine the block level of each variable so that it can generate the correct base address.

The parameter table is used to describe all parameters declared in the user program. This table is stored in the variable ParamTabl, which is an array of type ParamEntryType. This is a record type which references an enumerated type, ParmType. Each element in the array describes a single parameter. A variable, ParamTablPtr, contains the number of parameter descriptions stored in ParamTabl. The types ParmType and ParamEntryType are defined as follows:
type
ParmType = (ValPtype, VarPtype, RetPtype);

ParamEntryType = record
  TypePointer : integer;
  IsVar : ParmType;
  AddrOffs : longint;
  IdNum : integer;
end;

The field TypePointer contains the index number of the element within the type table that describes the data type of the parameter. The field IsVar is an enumerated type field. Its value is used to determine the classification of the parameter. As can be seen, there are three different classifications of parameters. Two of these, value and var, are found in standard Pascal. The third, ret, is an extension defined in BPL to designate process and resource procedure parameters which are returned to but not sent by the caller. When compiling calls to standard procedures and functions, only value and var parameters are permitted.

The IsVar field is used for several different tasks by the compiler. Whenever a var or ret parameter is being parsed, the compiler must accept only a variable of the appropriate type; value parameters, on the other hand, can be parsed as expressions. When accessing a procedure or function parameter, the compiler inspects the IsVar field when calculating the address of the parameter. If the value of this field is ValPtype, the address of the parameter is within the stack frame; otherwise, a pointer to the parameter is stored within the stack frame. When parsing a process or resource procedure call, the compiler inspects the IsVar field for each parameter. It then generates appropriate code to send the value of the...
parameter prior to the call, receive the value of the parameter after completion of the call, or both.

The AddrOffs field is used to indicate the location of the parameter within the stack frame. This is used in the same manner as the AddrOffset field used in the variable table to calculate the addresses of variables.

Finally, the field IdNum contains the unique integer assigned by the lexical analyzer to represent the identifier for the parameter. This field is used within a procedure, function, or process to access the parameters. It is not, however, accessed external to the procedure, function, or process since the name is not defined there.

4.3.2 Compiler Table Procedures

In the previous section, the tables maintained by the syntactic analyzer are described. Throughout the compilation of a BPL program, these tables are modified and examined by the various sections of the syntactic analyzer. In order to control these modifications and examinations, several housekeeping procedures are provided to perform these tasks. This is an advantageous approach to accessing these tables in that it allows the enforcement of rigorous modification policies as well as facilitating changes to the table data structures without significant changes to the code that accesses the tables.

The first compiler table procedure called by the compiler is InitTables. This is a very simple procedure that merely allocates memory for each table and sets
the table size variable to zero to indicate that each table is empty. As the compiler executes, these tables will be filled with data corresponding to the user's program.

Several procedures are used which merely add an entry to the appropriate table. Each of these procedures are passed several parameters. These parameters correspond to the fields of an element of the appropriate table. These table adding procedures perform essentially the same tasks. First, the table pointer variable for the appropriate table is incremented to point to the element for the new entry. Next, the table pointer variable is inspected to determine if the table is full. If the table is found to be full, the compiler halts and issues an appropriate error message. The parameters passed to the procedure are then copied into the element of the appropriate table array.

Such a procedure exists for each of the tables discussed above. The procedure StoreTypeData is used to store information regarding a type declaration into the type table. This procedure is called either when scanning type declarations or when scanning a variable declaration in which a new type is specified. The procedure StoreVarData is used to store information for a declared variable. This procedure is called once for each variable declaration encountered within the user program.

The procedures StoreFuncData and StoreProcData are used to store data in the function and procedure tables, respectively. These procedures are used each time such a program component is declared. The procedure StoreParamData is used to store information in the parameter table for each procedure or function
parameter. It is called once for each parameter encountered in a function or procedure declaration.

Another group of procedures are used to store data describing the different types of identifiers into the identifier table. As described above, the identifier table contains the unique identifier number of the identifier, the "type" of the identifier, the block level where the identifier is declared, and an extra long integer parameter. This long integer parameter represents different values for each different type of identifier.

The procedure DecConst is used to store a new element in the identifier table describing an identifier that represents a user-specified constant. The parameters of the procedure include the name of the constant identifier, the type of constant the identifier represents, and a corresponding long integer value. For integer, boolean, and character constants, the long integer represents the ordinal value of the constant quantity. For string and real valued constants, the long integer is an index into a table of string or real values, respectively, that points to the element represented by the constant identifier. This procedure is used each time a user-specified constant is encountered in the source code.

The procedure DecType is used to whenever a type specification is encountered in the type declaration section of a program. Note that, unlike type specifications in variable declaration sections, identifiers are assigned to all type specifications in a type declaration section. Two parameters are passed to DecType. The first parameter is the name of the identifier, and the second is a
pointer into the type table that indicates the element with data describing the type.

In a manner similar to that used by the DecType procedure, the procedures DecVar, DecProc, DecFunc, and DecParam are used to store the name and an associated element pointer for identifiers represent variables, procedures, functions, and parameters, respectively. The element pointer points into the variable table, the procedure table, the function table, or the parameter table, respectively, where the data for the particular identifier is stored.

The function AlreadyDef returns a boolean value indicating whether or not a specific identifier is already defined in a current block level. The number of the identifier is passed as a parameter to the function. The function returns a value of true is the identifier is already declared; otherwise, it returns a value of false. This function is used each time a new identifier is declared in order to enforce the naming rules of Pascal. Pascal states that, in a given block level, a given identifier cannot be declared more than once. For example, it is illegal to have a procedure and a variable in the same block level with the same name. Prior to declaring any type of identifier, the parser calls the AlreadyDef function. If a value of true is returned, the parser will issue an error message and terminate.

The procedure GetIdDesc is used to retrieve data from the identifier table regarding a specific identifier. Three parameters are passed to this procedure. The first is the unique integer corresponding to the identifier in question. The second parameter is a structured var parameter. GetIdDesc writes all pertinent data for the identifier into this structured variable. The third parameter, Found, is a
boolean var parameter. The procedure assigns a value of true to the Found parameter if the identifier is found in the identifier table. Otherwise, a value of false is assigned. The parser uses this procedure throughout the compilation in order to determine the characteristics of each identifier encountered. If, after calling the procedure, the Found parameter is returned with a value of false, the compiler issues an error message indicating that an identifier is not declared. The compiler then terminates.

The procedure StartProcessLevel is called whenever a process type is parsed. The block structure rules of BPL specify that no variables, procedures, or functions external to a process type can be referenced within the process type. The only external entities that can be referenced within a process are types and constants. The procedure GetIdDesc implements the enforcement of this rule in the following manner: When GetIdDesc finds the specified identifier in the identifier table, it checks to see if the block level of the identifier is less than the block level of the innermost process type being parsed. (The innermost process block level must be considered since processes can be nested.) If the block level is less than the block level of the innermost process, the identifier is external to the process. Therefore, unless the identifier is a type or constant identifier, GetIdDesc will respond as if the identifier does not exist.

In order to keep track of nested process levels, a first-in-first-out stack is implemented. Each process level is represented by a stack entry. If no processes are currently being parsed, the stack is empty. Otherwise, the block level of the
innermost process is found on top of the stack. The operation of the StartProcessLevel procedure is very simple. It merely pushes the current block level onto this stack.

A complimentary procedure named EndProcessLevel is used to terminate a process level. The operation of this procedure is to remove the top value from the first-in-first-out process level stack.

4.4 Code Generation

The code generation portion of the BPL compiler produces the executable machine code equivalent of the input BPL source program. A single procedure, Emit, is used as the interface to the code generator. Emit is called in order to produce executable code corresponding to one of the intermediate code instructions. Two parameters are used with the Emit procedure. The first parameter represents the intermediate code instruction to be expanded. The second parameter is a long integer value which is used to optionally convey additional information needed to generate code for the specific intermediate code instruction being expanded.

The model of code generated is that of Reverse Polish Notation, or RPN similar to that used in [Brode,1981] and [Bergmann,1986]. In an RPN model, the operands are first pushed onto an operand stack. Operators then examine the contents of the operand stack, perform some operation based upon these contents, and push the results back onto the operand stack. For example, in order to add
two numbers in an RPN environment, the two numbers are first pushed onto the operand stack. Next, an addition operator removes the two numbers from the stack, calculates the sum, and pushes the result back onto the stack.

The syntactical analyzer performs the task of transforming the input BPL program into an equivalent RPN expansion. This function is conveniently integrated into the recursive descent parser. The RPN expansion of the input program is expressed by a sequence of intermediate code instructions. An extensive group of intermediate code instructions are used in the BPL compiler. Each of these will be described shortly.

4.4.1 The BPL Compiler Virtual Stack

Unlike conventional RPN languages, the run-time code produced by the BPL compiler does not actually implement an operand stack. Instead, a virtual operand stack is maintained by the compiler at compile time. Conventional RPN implementations contain a memory-based last-in-first-out operand stack. This stack contains the operands used by the RPN program, including constants, calculated numeric values, logical flags, and address pointers. A pointer variable, called the virtual stack pointer, points to the top operand on the stack. The value of the virtual stack pointer changes at run-time when operands are added to or removed from the stack.

The virtual stack used within the BPL code generator is stored in the variable CStack, which is an array of type StackDataType:
**type**

StackDataType = record
  SType : (Empty, ConstOnly, RegOnly, ConstPlusReg, ConstTimesReg,
          StrOnly, CondC);
  ConstVal : double;  {value of possible constant}
  IsReal : boolean;
  CondField : byte;
  BaseAddr : (NoBase, VarAddr, ParamAddr);
  BLevel : byte;
end;

**const**

MaxSP = 7;  {maximum value of virtual stack pointer}
CondCC = 4;
CondCS = 5;
CondEQ = 7;
CondGE = 12;
CondGT = 14;
CondHI = 2;
CondLE = 15;
CondLS = 3;
CondLT = 13;
CondMI = 11;
CondNE = 6;
CondPL = 10;
CondVC = 8;
CondVS = 9;
CondT = 0;
CondF = 1;

**var**

SP : integer;  {the virtual stack pointer}
Cstack : array [0..MaxSP] of StackDataType;
  {the virtual stack}

The variable Cstack consists of eight elements of type StackDataType. Therefore, the virtual stack can represent a maximum of eight operands. This is sufficient even for very complex instructions. The variable SP is the virtual stack pointer. When the virtual operand stack is empty, SP contains a value of negative
one. When the virtual operand stack contains one operand, the value of SP is zero, and so on.

The type StackDataType provides a flexible representation for many possible type of operands. The field Stype indicates the general form of the operand. An operand with an SType value of Empty indicates that no value is stored in the operand. An SType value of ConstOnly indicates that the operand contains an integer or real constant. This constant value is stored in the field ConstVal. An SType value of RegOnly indicates that the operand is a value stored in a data register. The actual data register used corresponds to the element number within the CStack variable array. For example, an operand with the field SType having a value of RegOnly contained in element zero of CStack would represent an integer value stored in data register D0 or a real value stored in floating point register FP0.

An entry with SType of value ConstPlusReg designates an operand whose value is equal to a constant value added to a value stored in a register. An entry with SType of value ConstTimesReg designates an operand whose value is equal to a constant value multiplied by a value stored in a register. In both of these cases the constant value is stored in the field ConstVal. The register used for the addition or multiplication is the same register as described above for RegOnly operands.

An operand with an SType value of StrOnly represents a character string operand. Character strings in BPL are 256 bytes in length, consisting of a byte
value indicating the length of the string and 255 bytes containing each character of the string. Strings less than 255 characters in length contain indeterminate data in the unused character locations. Intermediate string calculations obviously cannot be stored in the registers of the processor. Therefore, they are stored by allocating space on the 680x0 microprocessor's return stack. Therefore, when an operand with a value of StrOnly assigned to the SType field is encountered, the compiler assumes that the 680x0 stack space is adjusted to allocate space for the string.

An SType field with a value of CondC indicates that the operand is a boolean value stored in the condition code registers of the 680x0 microprocessor. Motorola defines sixteen possible condition code combinations, with a four bit binary number assigned to each combination. Each of these condition values is defined above as a constant. The field CondField contains the actual condition code represented by the operand. For example, consider the situation where two operands are tested for equality. Following the comparison, the top operand on the virtual operand stack would be of type CondC. The field CondField would have a value of CondEQ. This operand represents a boolean result whose value is true if the condition code register of the 680x0 microprocessor is set such that the equality condition is true.

As mentioned above, the field ConstVal is used to store a constant value for operands of type ConstOnly, ConstPlusReg, or ConstTimesReg. For other types of operands, the value of the ConstVal field is undefined.
The field IsReal is a boolean variable used to designate between integer and floating point operands. When this field is true, and value stored in the field ConstVal represents a real value. Any value stored in a register is stored in a floating point register of the 68881 floating point coprocessor. Conversely, when IsReal is false, the value stored in the field ConstVal is an integer value, and any register value is stored in a data register of the 680x0 microprocessor.

The field BaseAddr indicates if a base address value should be added to the operand value. This field is used if the operand is a variable or parameter address. In the code model used for BPL, variables and parameters for a given block level are stored contiguously in memory beginning at a base address. For example, the first variable declared in a given block level is stored at the base address for variables of that block level. The second variable declared for that block level immediately follows the first variable in memory. Therefore, in order to calculate the address of a variable, the offset within the appropriate block level must be added to the base address for variables of that level. Within the virtual operand stack, variable and parameter addresses are designated by the BaseAddr field. The three possible values for this field, NoBase, VarAddr, and ParamAddr, indicate that no base address is added, the variable base address is added, or that the parameter base address is added, respectively.

This mechanism is useful in that it allows use of many of the 680x0’s addressing modes when referencing variables. For example, a simple variable will be stored at a constant offset from the base variable address. Therefore, the
address of such a variable would be represented by an integer operand of type ConstOnly with a BaseAddr of VarAddr. The base address for variables is stored in a 680x0 address register. Therefore, the variable can be referenced in one instruction by using address register indirect with displacement mode.

The BLevel field is used to designate the block level of a variable or parameter access. BPL implements the block nesting of standard Pascal. Therefore, nested procedures and functions can access the variables and parameters of parent procedures, as well as the global variables of the program. Each different block level's variable and parameter base addresses are linked to those at the lowest level. Therefore, the code generator must keep track of the block level of each variable access so that the appropriate base address is used.

4.4.2 Run-Time Conventions

Several of the 680x0 address registers permit specific tasks during the execution of a BPL program. These registers, and their associated tasks, are as follows:

A7 : Return stack pointer for procedure and function calls; pointer for intermediate string calculation results

A6 : Base pointer for global variables
A5 : Base pointer for nested procedures. The variables for the current
block level begin at A5 + 4. The value stored in @A5 is the value
for A5 in the next parent block. Variables for a parent block
therefore begin at @A5 + 4. Variables for a grandparent block
begin at @@A5 + 4, and so on.

A4 : Base pointer for procedure and function parameters. The parameters
for the current block level begin at A4. The value stored in @(A4 -
4) is the value for A4 in the next parent block. Parameters for a
parent block therefore begin at @(A4 - 4). Parameters for a
grandparent block begin at @@(A4 - 4), and so on. A function
result is stored immediately following the parameters.

In order to maintain the required linkage between nested procedures and
functions, calling conventions are required. When calling a procedure or function,
the following operations are performed:

* If this is a function call, reserve space on the return stack for the result.

* Push any parameters onto the return stack.

* Execute the following instructions in order to have a4 point to the new
parameters with the old parameter pointer link stored:

  movea.l a4,a3  *save current param pointer
  movea.l a7,a4  *store new param pointer
  move.l a3,-(a7)  *save old param pointer

* Perform a call to the appropriate subroutine corresponding to the
function or procedure.

* Restore the parameter by executing the following instruction:

  movea.l (a7)+,a4

* Remove the parameters from the stack by executing the following
instruction:

  adda.l #ParamSize,a7
* If this is a function, get the result off the processor stack.

As can be seen, the above calling convention properly saves the old value of the parameter base pointer, loads the parameter base pointer with the correct new value, calls the procedure or function, and restores the parameter base pointer.

In addition to properly saving and restoring the parameter base pointer, the variable base pointer must be saved, recalculated for the new block level, and restored at the completion of the block level. These tasks are performed within the procedure or function being called in the following manner:

* Allocate space for local variables on the processor's return stack by executing the following instruction:

  suba.l #VariableSize,a7

* Save the old variable pointer by executing the following instruction:

  move.l a5,-(a7)

* Set the variable pointer to point to the current block level's variables by executing the following instruction:

  movea.l a7,a5

* Execute the instructions associated with the procedure or function.

* Restore the variable pointer to the previous value by executing the following instruction:

  movea.l (a7)+,a5
* Release the space allocated for local variables by executing the following instruction:

    adda.l #VariableSize,a7

* Return to the calling procedure by executing the following instruction:

    rts

This convention for procedure and function code ensures that the proper value for the variable base is calculated and that the outer block level variable linkage is correct.

When the program begins execution, the first instructions are used to allocate space for global variables and establish the correct global variable pointer:

    suba.l #GlobalVariableSize,a7
    movea.l a7,a6

As these code generation conventions demonstrate, all variable and parameter memory space is allocated from the 680x0 return stack. The operating system initializes the return stack pointer prior to execution of the program with a value equal to the total stack size plus the size of all global variables.

One calling convention remains: system calls to the operating system. All calls to the operating system are performed by executing a TRAP #2 instruction. Prior to executing the trap, the program loads calling parameters into the 680x0 data and address registers. All functions require a unique value stored in data register 0 used to indicate the operating system function to be performed. Other registers optionally contain data values, address values, and other calling
parameters specific to each operating system function. These functions will each be described in Chapter V.

4.4.3 Intermediate Code Operations Description

This section will describe each operation in the intermediate code definition. As each instruction is described, a discussion of the generated code and the changes to the virtual stack will be presented.

As mentioned previously, two operands are passed when calling the Emit procedure. The first parameter is the name of the intermediate code instruction being invoked. The second parameter is an integer value which is used to pass additional information specific to each intermediate code instruction. The name of this integer value parameter is Param; it will be referred to extensively in the following sections.

The intermediate code instructions can be conveniently grouped according to function. The following discussion will be similarly organized.

4.4.3.1 Program Control Operations

DecLabel: The DecLabel instruction is used to insert an address label into the assembly language code at the current position. Param is the integer number of the label to be inserted. All labels have the form Ln, where n is a positive integer. Therefore, if a DecLabel instruction were called with Param having a value of seven, the result label would be L7.
**JmpTrue** : JmpTrue is used to generate a branch instruction which will execute only if the boolean value stored on top of the virtual stack is true. Param is the label number that is used as the destination of the branch. The code generated by this instruction can take on several different forms. If the top operand on the virtual stack is a condition code, a single Bcc instruction is generated, where cc is the top operand condition code value. If the operand is a constant, the value of the constant is inspected; a non-zero constant will result in a branch always instruction, while a zero constant will result in no instruction. For other operand types, the result is reduced and stored in the appropriate register. This involves generating instructions to add or multiply any constants and add any variable or parameter offsets. Code is generated to test the resulting register and branch if the result is non-zero. The top operand is removed from the virtual stack at the completion of this operation.

**JmpFalse** : JmpFalse is similar to JmpTrue except that the branch statement is executed if the top operand is evaluated as false. For the situation where the top operand is a condition code, the negative condition code is determined and a single branch instruction is generated using that code. Otherwise, the procedure is identical to JmpTrue except that the branch is performed if a zero result is obtained. The top operand is removed from the virtual stack at the completion of this operation.

**JmpAlways** : JmpAlways causes a branch always instruction to be generated. Param contains the label number that is used as the destination of the
branch.

**ProgStart** : ProgStart generates code used to allocate space for the global variables. As described above, this involves subtracting the size of the global variables from the processor return stack pointer. The new value of the processor return stack pointer is copied into address register six, which is the base address pointer for global variables.

**ProgEnd** : ProgEnd is used to generate code to terminate a program and return control to the operating system. This is done by loading a zero into data register zero and executing a TRAP #2 instruction.

**ProcStart, FuncStart** : ProcStart and FuncStart are identical operations that generate the entry code for procedures and functions. As described above, these instructions allocate space for local variables on the processor return stack, save the previous variable base pointer, and initialize the new variable base pointer. Param is the size, in bytes, of the local variables for the procedure or function, which is the amount of memory allocated from the processor return stack.

**ProcEnd, FuncEnd** : ProcEnd and FuncEnd are identical operations that generate the exit code for procedures and functions. As described above, these instructions restore the value of the previous variable base pointer, release any local variable space, and return to the calling procedure. Param is the size, in bytes, of the local variables for the procedure or function, which is the amount of memory released from the processor return stack.

**ProcCall** : ProcCall is used to generate the code for calling a procedure.
Param is a pointer into the variable ProcTabl where data is stored for the procedure to be called. First, the procedure’s block level is determined from the variable ProcTabl. From this value, the base addresses of the parameters and variables for the parent level are calculated and properly linked. Next, a branch to subroutine instruction is generated which calls the subroutine associated with the procedure. The label number for this subroutine is stored in the variable ProcTabl. Upon return from the subroutine, code is generated to restore the parameter and variable base pointers to the current level. Next, the variable ProcTabl is inspected to determine the size of the parameters. This amount is added to the processor stack pointer in order to remove the parameters from the stack.

**FuncCall**: FuncCall is used to generate the code for calling a function. Param is a pointer into the variable FuncTabl where data is stored for the function to be called. First, the function’s block level is determined from the variable FuncTabl. From this value, the base addresses of the parameters and variables for the parent level are calculated and properly linked. Next, a branch to subroutine instruction is generated which calls the subroutine associated with the function. The label number for this subroutine is stored in the variable FuncTabl. Upon return from the subroutine, code is generated to restore the parameter and variable base pointers to the current level. Next, the variable FuncTabl is inspected to determine the size of the parameters. This amount is added to the processor stack pointer in order to remove the parameters from the stack. Finally the function
result is removed from the stack. Again, the variable FuncTabl is inspected to
determine the result type of the function. Because of the restrictions of Pascal, this
result type will be either an integer value, a floating point value, or a string.
(Note that an integer value can also represent boolean or enumerated types.) The
virtual stack pointer, SP, is incremented and the result type information is stored
in the top operand of the virtual stack. Because of the type restrictions, the field
SType will be either RegOnly or StrOnly. If the result type is floating point, the
field IsReal will be set to true. For string type results, no further action is taken,
since string results are stored on the processor return stack. For integer results,
the integer value is removed from the stack and stored in the appropriate data
register. The number of the data register used is the number stored in the virtual
stack pointer, SP. If the result type is floating point, the value is removed from
the stack and stored in the appropriate floating point register.

PrepFuncCall: PrepFuncCall is used to generate code for reserving space on
the processor return stack for a function result. Param is a pointer into the
variable FuncTabl where data is stored for the function to be called. This
operation inspects FuncTabl to determine the type of the function's result. It then
adjusts the processor stack to hold a variable of that size.

4.4.3.2 Case Statement and For Statement Operations

EvalCase: EvalCase is used to reduce the case expression and move the
result into data register zero. This operation is performed just before the first case
comparison is made. Throughout all case comparisons, the case expression value remains in data register zero. It remains there either until all case constants have been compared and no matches were found, or until a match is found and a corresponding statement is executed. The value placed into data register zero is removed from the top of the virtual operand stack.

**CaseCmp** : This operation expects to find a constant value on the top of the virtual operand stack. This constant value is compared to the value stored in data register zero. If a match occurs, a branch is made to execute the corresponding statement. The top operand on the virtual stack is removed.

**ForPrelim** : This operation is used to prepare the processor return stack for the execution of a for statement. Three operands are expected on the virtual operand stack. The top operand is the final value of the control variable. The second operand is the initial value of the control variable. The third operand is the address of the control variable. Param is a pointer into the variable TypeTabl which points to the description of the type of the control variable. This operation generates several instructions. The first instruction stores the initial value of the control register into the control register. Next, the address of the control register is pushed onto the processor return stack. The top operand on the virtual stack is now subtracted from the second operand. The difference is pushed onto the processor return stack.

**ToCmp, DowntoCmp** : These operations are used to generate code executed before the first iteration of a for statement. The resulting code checks to see if the
initial control variable value is greater than or less than the final control variable value for for/to and for/downto statements, respectively. This is done by inspecting the value on top of the processor return stack, which is the difference of the initial value and the final value. Param is the label number of the clean-up code for the for statement; jumping to this label will skip over the iterative portion of the for statement. ToCmp causes a jump to the clean-up code if the value on top of the stack is positive. DowntoCmp causes a jump to the clean-up code if the value on top of the stack is negative.

**ForInc**: This operation is called following the execution of the statement portion of a for statement using the to directive (as opposed to the downto directive.) Prior to the execution of the iterated statement, code produced by the ForPrelim operation pushed two values onto the stack. The top value is the difference between the initial and final values of the control variable. The second value is the address of the control variable. ForInc first increments the control variable pointed to by the second value on the stack. Param is a pointer into the variable TypeTabl which contains information regarding the type of the variable to be incremented. Next, ForInc increments the top value on the processor return stack. The result of this increment is reflected as a condition code value on top of the virtual operand stack. The condition code for this is CondLE, meaning that the condition is true if the resulting value is less than or equal to zero. The syntactic analyzer will always immediately follow this operation with a JmpTrue operation back to the beginning of the iterated statement. Therefore, if the result
of incrementing the value on top of the processor return stack is greater than zero, the for statement will not repeat again.

**ForDec**: This operation is called following the execution of the statement portion of a for statement using the `downto` directive (as opposed to the `to` directive.) Prior to the execution of the iterated statement, code produced by the `ForPrelim` operation pushed two values onto the stack. The top value is the difference between the initial and final values of the control variable. The second value is the address of the control variable. `ForDec` first decrements the control variable pointed to by the second value on the stack. `Param` is a pointer into the variable `TypeTabl` which contains information regarding the type of the variable to be incremented. Next, `ForInc` decrements the top value on the processor return stack. The result of this decrement is reflected as a condition code value on top of the virtual operand stack. The condition code for this is `CondGE`, meaning that the condition is true if the resulting value is greater than or equal to zero. The syntactic analyzer will always immediately follow this operation with a `JmpTrue` operation back to the beginning of the iterated statement. Therefore, if the result of incrementing the value on top of the processor return stack is less than zero, the for statement will not repeat again.

**ForEnd**: `ForEnd` is used to produce clean-up code at the completion of a for statement. The code produced by this operation simply removes the two values placed by `ForPrelim` from the processor return stack.

**EnablePar, DisablePar**: BPL introduces an extension to the for statement
of standard Pascal. The extension provides for the use of a for statement within a parallel statement in order to simultaneously initiate an array of processes. The basic code produced by each form of the for statement is similar. The parallel form, however, incorporates the parsing of a process call rather than a statement. When parsing a process call, a long integer value must be found on top of the processor return stack. The for statement, however, pushes two values on top of the processor return stack. Therefore, for the parallel form of the for statement, the ForPrelim first removes the long integer from the top of the processor return stack, pushes its two values onto the stack, and then returns the removed value onto the stack so that the parsing of the process call will work properly. When performing the ForInc and ForDec operations in the parallel form, the long words on top of the processor return stack for the standard for statement are now the second and third long words on the processor return stack. Finally, the ForEnd operation, which ordinarily discards the top two long words from the processor stack, removes the long integer from the top of the processor stack, discards the next two long words, and then returns the long integer to the stack.

A boolean variable, ParEnabled, is set to true if the parallel form of these operations is to be used. Otherwise, the standard form is used. The EnablePar and DisablePar operations simply set the boolean variable ParEnabled to true or false, respectively.

4.4.3.3 Memory Operations
PushBaseVarAddr, PushBaseParamAddr: These operations produce no code. They simply push a new operand onto the virtual operand stack. The SType field for this operand is Empty. The BaseAddr field for this operand is VarAddr or ParamAddr, respectively. The BLevel field is set to the value passed in Param, which is the block level in which the variable of parameter being accessed is declared. The operand created will most likely be added to another operand containing, for example, a constant value in order to produce the address of a simple variable.

PushStruct: This operation is used to push a structured variable onto the processor return stack as a value parameter. Because of standard Pascal restrictions, the result of any expression must be either an integer value, a floating point value, an enumerated value, a boolean value, a character value, or a string value. Therefore, any expression for a structured value parameter must consist of a single variable of the same type. This operation is used to copy that structured variable onto the stack. The top operand on the virtual operand stack is the address of the variable to be copied onto the stack. Param contains the size, in bytes, of the structured type. After generating the code to copy the variable, PushStruct removes the top operand from the virtual operand stack.

CopyStruct: Because of standard Pascal restrictions, the result of any calculation must be either an integer value, a floating point value, an enumerated value, a boolean value, a character value, or a string value. Therefore, any assignment to a variable such as a record or array type consists only of copying
another variable with an identical type. CopyStruct is used to generate the code to perform the copy operation. CopyStruct uses the top two operands on the virtual operand stack. The top operand is the source address, while the second operand is the destination address. Param contains the size, in bytes, of the structure to be copied. After generating the appropriate code, CopyStruct removes the top two values from the virtual operand stack.

**PushVal** : PushVal is used to generate code to push the result of a calculation as a var parameter onto the processor return stack. The result of the calculation is stored in the top operand on the virtual stack. Param contains a pointer into the variable TypeTabl where data regarding the type of the var parameter being stored can be found. Because of standard Pascal restrictions, the result of any calculation must be either an integer value, a floating point value, an enumerated value, a boolean value, a character value, or a string value. These types reduce to integer, real, or string in the intermediate code language. Integer and real values are reduced, if necessary, with the result begin a data or floating point register numbered by the virtual stack pointer. String values will be found on the processor return stack. For integer and real values, the contents of the appropriate register are pushed onto the stack. For string values, the string is already on top of the processor return stack, and therefore no action is required. At the conclusion of this operation, the top operand is removed from the virtual stack.

**PushVar** : This operation is used to generate code to push a var parameter
onto the processor return stack. Var parameters are pushed by pushing the address of the parameter variable. The address of this variable will be found in the top operand on the virtual stack. This top operand will then be discarded from the operand stack.

**Fetch**: Fetch is used to generate code to read from a variable a value to be used in a calculation. Param contains a pointer into the variable TypeTabl where data regarding the type of the variable being fetched can be found. The address of the variable to be fetched is represented by the top operand on the virtual operand stack. This operation discards the memory address from the top of the virtual operand stack. The value read from memory will be stored on top of the virtual operand stack. Because of standard Pascal restrictions, any value used in a calculation must be either an integer value, a floating point value, an enumerated value, a boolean value, a character value, or a string value. These types reduce to integer, real, or string in the intermediate code language. For integer or real values, the results will be stored in the appropriate data or floating point register and the top operand on the virtual operand stack with have a value of RegOnly for the SType field. For a string value, the string will be placed on top of the processor return stack and the SType field for the operand will have a value of StrOnly.

All integer values fetched from memory are converted to long integers, regardless of the original operand size. For byte and word operands, the upper bits are zeroed. For integer and shortint operands, the upper bits are signed
extended.

**Store** : Store is used to generate code to store the result of a calculation. The result of the calculation is stored in the top operand on the virtual stack. Param contains a pointer into the variable TypeTabl where data regarding the type of the variable being stored can be found. Because of standard Pascal restrictions, the result of any calculation must be either an integer value, a floating point value, an enumerated value, a boolean value, a character value, or a string value. These types reduce to integer, real, or string in the intermediate code language. Integer and real values are reduced, if necessary, with the result begin a data or floating point register numbered by the virtual stack pointer. String values will be found on the processor return stack. The second operand on the virtual stack is the address where the result is to be stored. For integer and real values, the contents of the appropriate register are stored in the memory address. For string values, the string is copied from the top of the processor return stack to the memory address. At the conclusion of this operation, the top two operands are removed from the virtual stack.

### 4.4.3.4 Math Operations

**IntConstant, BoolConstant, RealConstant** : These operations are used to push constant values onto the virtual operand stack. The result of these operations is a new operand with the SType field having a value of ConstOnly. The field IsReal is set to true for the operation RealConstant, while the IntConstant
and BoolConstant operations set it to false. No executable machine code is produced by these operations; they merely add an entry to the virtual operand stack. For the IntConstant and BoolConstant operations, Param contains the actual integer or boolean value, with zero representing logical false and one representing logical true. For the RealConstant operation, Param contains an index into a table of real constants and points to the real constant to be used.

Add: The Add operation generates any code instructions needed to add the top two operands on the virtual operand stack. The top two operands are removed from the virtual operand stack and the result is pushed back on. This operation does not always generate instructions. For example, if the top two operands on the virtual operand stack were constant values, the sum of the constants is calculated by the compiler and the resulting constant is pushed back to the stack. Another possible situation is where one operand is a register value and the other is a constant. The resulting operand will have an SType of ConstPlusReg. No code is needed to make this transformation. If either of the top two operands prior to the Add operation is a floating point type, the result will be a floating point type. Otherwise, the result will be an integer type.

Subtract: The Subtract operation generates any code instructions needed to subtract the top operand from the second operand on the virtual operand stack. The top two operands are removed from the virtual operand stack and the result is pushed back on. As with the Add operation, this operation does not always produce executable code. Any possible simplifications which can be made are
performed at compile time. If either of the top two operands prior to the Subtract operation is a floating point type, the result will be a floating point type. Otherwise, the result will be an integer type.

**Mult**: The Mult operations is used to generate any code instructions needed to calculate the product of the top two operands on the virtual operand stack. The two operands are removed from the virtual operand stack and the result operand is pushed back on. As described previously for other math operations, any possible simplifications will be performed at compile time; therefore, this operation will not necessarily produce executable code. If either of the top two operands prior to the Mult operation is a floating point type, the result will be a floating point type. Otherwise, the result will be an integer type.

**Divide**: This operation is used to generate any instructions needed to divide the second operand on the virtual operand stack by the top operand on the virtual operand stack. This operation removes the top two operands from the virtual operand stack and pushes back the result. The result of this operation is always a real value regardless of the operands used. As described previously for other math operations, any possible simplifications will be performed at compile time; therefore, this operation will not necessarily produce executable code. In particular, if the top operand on the virtual operand stack is a constant and the second operand is a constant, the quotient is calculated at compile-time and the result pushed as a real constant operand onto the virtual operand stack. If the top operand is a constant and the second operand is a stored in a register, the result
is a real operand with an SType field of value ConstTimesReg, where the constant is equal to the inverse of the constant operand on top of the virtual operand stack.

**IntDiv**: This operation is similar to the Divide operation in that it produces any code needed to divide the second operand on the virtual operand stack by the top operand. The distinction is that both operands must be integer values and that the quotient is an integer value with the fractional portion of the result truncated. This operation is used in conjunction with the `div` operator in the Pascal language. The only situation where compile-time simplifications can be made is where both operands on the virtual operand stack are constants. In this situation, the constant result is calculated by the compiler. Otherwise, the compiler generates the necessary division instruction to obtain the quotient.

**Modulo**: The Modulo operation is used to produce any code needed to divide the second operand on the virtual operand stack by the top operand and obtain the integer remainder of the result. Like IntDiv, both operands must be integer. The only situation where compile-time simplifications can be made is where both operands on the virtual operand stack are constants. In this situation, the constant result is calculated by the compiler. Otherwise, the compiler generates the necessary division instruction to obtain the remainder.

**AddConstant, MultConstant**: These operations are merely combinations of previous operations. Their purpose is to add or multiply an integer constant, respectively. They are primarily used when calculating variable addresses. For example, each variable is stored at a constant offset from a predefined base address.
register. Therefore, a variable address can be obtained by issuing a PushBaseAddress operation followed by an AddConstant operation. MultConstant is useful when working with arrays. Each time an array index is evaluated, it must be multiplied by a constant corresponding to the stride of the array. For both of these operations, Param contains the integer value to be used.

**DoSin, DoCos, DoTan, DoACos, DoASin, DoATan, DoExp, DoLn, DoSqrt**:
These operations are used to generate code to perform a variety of mathematical functions on the top operand on the virtual operand stack. The first step of each of these operations involves simplifying the top operand on the virtual operand stack and transferring the result to the floating point register numbered by virtual stack pointer, SP. Then, a single floating point instruction acting upon the floating point register is generated to perform the appropriate function, leaving the result in the same register. All of these operations leave one real, register contained value on top of the virtual operand stack.

**ChkRange**:
ChkRange is used to generate code that inspects the value of expressions and determines that they are within bounds for a specified type. Param contains a pointer into the variable TypeTabl where data regarding the type of the expression being checked can be found. This operation is used for assignments to variables and for checking the value of an array index expression. The generated code forces a TRAPcc instruction if the value is found to be out of range.

Range checking requires additional generated instructions, and therefore
slows down the execution of a BPL program. The programmer is given the option to turn range checking on and off by imbedding "option comments" within the source code. To enable range checking, the comment {$R+} is included in the program, while {$R-} is included to disable the feature. Range checking can be enabled and disabled repeatedly throughout a program, allowing the trapping of errors in selected code regions. The default condition for range checking off.

**IncVar, DecVar**: These operations are used to generate code for the Inc and Dec procedures of Pascal. Inc and Dec provide an efficient mechanism for incrementing and decrementing scalar variables. The top operand on the virtual operand stack contains the address where the variable to be incremented or decremented is stored. Param contains a pointer into the variable TypeTabl where data regarding the type of the variable being incremented or decremented can be found. These operations generate the 680x0 ADDQ and SUBQ instructions. The operand size is determined by the type definition pointed to by Param.

### 4.4.3.5 String Handling Operations

**CmpStrs**: This operation is used to generate the necessary code to compare two string values. The two strings represent the top two operands on the virtual operand stack. After the completion of this operation, both operands are removed from the operand stack and the boolean result is pushed.

The strings to be compared are stored at run-time on the processor return stack. They are stored in consecutive blocks of memory. Following the code that
performs the comparison, code is produced that increments the processor stack
pointer, discarding the strings.

The value of Param is used to indicate the comparison to be performed. The values one through six correspond to the comparisons <, <=, =, >, >=, and >. If the indicated comparison is found to be true, a value of one is pushed onto the virtual operand stack. Otherwise, a value of zero is pushed.

CmpStrChar, CmpCharStr : These operations are used to compare two strings where one of the strings is a single character. CmpCharStr and CmpStrChar are used when the first and second operand on the virtual operand stack is a character, respectively, and the other operand is a string. The string being compared is stored on the processor return stack, while the character is stored either in a register or as a constant. Following the production of the code to perform the actual comparison, code is produced to remove the string being compared from the processor return stack.

The value of Param is used to indicate the comparison to be performed. The values one through six correspond to the comparisons <, <=, =, >, >=, and >. If the indicated comparison is found to be true, a value of one is pushed onto the virtual operand stack. Otherwise, a value of zero is pushed.

StrConstant : This operation is used to generate code that moves a string constant into space allocated on the processor return stack. Additionally, this operation indicates the presence of a string on the stack by pushing a string operand onto the virtual operand stack.
The value of Param is an index into a table of string constants. The compiler stores all string constants in this table. The produced code first subtracts the required amount of space from the processor return stack pointer. The actual character values corresponding to the appropriate string are then copied into the newly allocated space.

**StoreStr** : This operation is used to produce the code to store the result of a string expression into a string variable. The string is the top operand on the virtual operand stack, while the address of the string variable is the second operand. Because the storage of a string is a lengthy process, a subroutine is included at the end of the program to perform the operation. The compiler, however, only includes this subroutine if it is needed. Therefore, this operation first sets a flag to indicate that the subroutine is needed. It then executes a call to the appropriate subroutine. Finally, the string and the string variable address are removed from the virtual operand stack.

**PushStrValParam** : Due to the code model for strings in BPL, string results are stored on the stack in a identical manner to string value parameters. Therefore, this operation is not needed to produce any executable code. It is needed, however, to remove the string operand from the top of the virtual operand stack.

**FetchStr** : This operation is used to produce the code needed to fetch a string variable's value from memory and copy it into a block of memory reserved on the processor return stack space. The top operand on the virtual operand stack
is the address of the string variable. Because the fetching of a string is a lengthy process, a subroutine is included at the end of the program to perform the operation. The compiler, however, only includes this subroutine if it is needed. Therefore, this operation first sets a flag to indicate that the subroutine is needed. It then executes a call to the appropriate subroutine. Finally, the string variable address is removed from the virtual operand stack and a string operand is pushed.

**PushCharAsStrParam** : BPL allows characters to be used as single-character strings. This operation is used when such a single-character string is used as a value parameter in a procedure or function call. This operation creates the code needed to allocate space from the processor return stack and store the single character in the allocated space in standard string format. The character operand is then removed from the virtual operand stack.

**StoreCharInStr** : BPL allows characters to be used as single-character strings. This operation is used when such a single-character string is assigned to a string variable. The top operand on the virtual operand stack is the character, and the second operand is the address of the string variable where the character is to be stored as a string. The compiler produces the code to store the character as a string of length one. The character and address operands are then removed from the virtual operand stack.

**AddStrings** : This operation is used to produce code to concatenate two strings. The two strings to be concatenated are the first two operands on the virtual operand stack. The actual strings are store at run-time on the processor
return stack. Because of the length of the code needed to perform this task, a subroutine is used to perform the concatenation of the two strings. The compiler, however, only includes this subroutine if it is needed. Therefore, this operation first sets a flag to indicate that the subroutine is needed. It then executes a call to the appropriate subroutine. Finally, the two string operands are removed from the virtual operand stack and the resulting string operand is pushed.

AddCharStr, AddStrChar: These operations are used to produce code to concatenate two operands into a string. One of the two operands is a character value, while the other is a string. For the AddCharStr operation, the top operand on the virtual operand stack is a character and the second operand is a string. For the AddStrChar operation, the top operand on the virtual operand stack is a string and the second operand is a character. Because of the length of code required to perform these tasks, the operations merely call a subroutine to perform the appropriate task. The compiler, however, only includes this subroutine if it is needed. Therefore, this operation first sets a flag to indicate that the subroutine is needed. It then executes a call to the appropriate subroutine. Finally, the string and character operands are removed from the virtual operand stack and a string operand is pushed.

AddCharChar: This operation is used to concatenate two character values into a two character string. The characters to be concatenated are the top two operands on the virtual operand stack. The produced code first allocates space on the processor return stack for the resulting string. Next, a length byte of two is
stored in the string, indicating that the resulting string will be two characters in length. The two characters are then moved into the string storage space. Finally, the two character operands are removed from the virtual operand stack and the resulting string operand is pushed.

**InitStr**: This operation is not used to produce any code. Its only purpose is to set to false all flags mentioned in the above string operation descriptions. **InitStr** is called at the very beginning of a program. At the end of a program, **CloseStr** is called, which inspects the status of the flags.

**CloseStr**: **CloseStr** is called at the end of a program's compilation in order to determine if any string function subroutines are needed. This is determined by inspecting the values of several status flags. Each flag corresponds to a string function subroutine. If the flag is set to true, the code for the corresponding subroutine is added to the program. In this manner, the only subroutines included are those actually called by the program.

### 4.4.3.6 Process and Resource Calling Operations

**ParallelCall**: This operation is used to generate the executable code that initiates a parallel control structure. During the parsing of the multiple process invocations contained within a parallel statement, the generated code places the address of the current process variable on the processor return stack. When subsequent process invocations are parsed, code is generated to remove this address from the stack, store it as a link within the current process variable being
parsed, and push the address of the current process variable onto the processor return stack. After parsing the final process invocation within a parallel statement, the address of the corresponding last process variable is on the stack. Within this process variable is stored a link to the previous process variable. This continues until a process variable is encountered with a link value of minus one. This represents the last process variable in the linked list, which is the process variable used in the first process invocation within the parallel statement.

The code generated by the ParallelCall operation performs two simple tasks. First, the address on top of the processor return stack is removed and stored in address register zero. Next, a value is loaded into data register zero which corresponds to the appropriate operating system function for invoking the parallel processes. Finally, a TRAP #2 instruction is generated, which causes a call to the operating system. The operating system will invoke the processes using the linked list pointed to by address register zero. Control will return to the calling program after the termination of all specified processes.

CallRsreProc : This operation generates code to call a resource procedure. Prior to calling this operation, the parser calculates the address of the resource variable to be used during the resource procedure's execution. This address is stored as the top operand on the virtual operand stack. The value of this address is removed from the virtual operand stack and stored in address register zero. A constant value corresponding to the appropriate operating system function is loaded into data register zero. Finally, a TRAP #2 instruction is used to transfer
control to the operating system. The operating system will suspend the calling process until the resource variable is available for exclusive use. The operating system will then execute a call to the appropriate procedure address. Finally, after the procedure call returns, the operating system returns control to the previously executing process.

StartProcess: This operation is used to generate code at the beginning of each process. Param contains the size of the variables local to the process. The generated code first allocates space on the processor return stack for the process local variables. Then, the value of the process return stack pointer is copied into address register six, allowing access to the process local variables.

EndProcess: This operation is used to generate the code at the completion of a process. The code generated by this operation is very simple. It merely loads a value into data register zero that represents the end of process operating system function. It then executes a TRAP #2 instruction which relinquishes control to the operating system.

StartRsrcProc: This operation is used to generate code at the beginning of each procedure declared in the interface section of a resource type declaration. Param contains the size of the variables local to the resource procedure. The generated code first allocates space on the processor return stack for the local variables. Next, the current value of address register five, which points to the current local variables, is pushed onto the processor return stack. Finally, the value of the process return stack pointer is copied into address register five,
allowing access to the procedure's local variables.

**EndRsreProc** : This operation is used to generate the code at the completion of a procedure declared in the interface section of a resource type declaration. The code generated by this operation is very simple. It merely loads a value into data register zero that represents the end of resource procedure operating system function. It then executes a TRAP #2 instruction which relinquishes control to the operating system.

### 4.4.3.7 Logical Operations

**LogAnd, LogOr, LogXor** : These operations are used to generate any code necessary to perform the appropriate binary logical operation. These operations remove the two top operands from the top of the virtual operand stack, calculate the bit-wise ANDing, ORing, or XORing of the two operands, and push the result back onto the virtual operand stack. As with other operand stack referencing operations, these operations will produce no code if the two top operands on the virtual operand stack are constants. Instead, the appropriate constant result will be calculated and pushed onto the virtual operand stack.

The operands used for this operation can either be integer or boolean values. Both, however, must be of the same type. The reason the same operation can be used for both types of operands is that boolean values are represented as integers, with a value of zero representing false and a value of one representing true. These values can be ANDed, ORed, or XORed in the same manner as integer
values, yielding the correct boolean result.

**LogNot, BitwiseNot** : These operations are used to calculate the result of the unary not operator of Pascal. Pascal allows the unary not to be used with both boolean and integer operands. The code produced for the two different types of operands, however, must be different. The unary not operator applied to a boolean operand transforms a value of zero to one and transforms a value of one to zero. The code for this situation is generated by the LogNot operation. The unary not operator applied to an integer operand compliments each bit of the operand. The code for this situation is generated by the BitwiseNot operation. In either case, the top operand is removed from the virtual operand stack. The appropriate code is generated to calculated the desired result. Data regarding this result is pushed back onto the virtual operand stack.

**CmpEQ, CmpNE, CmpLT, CmpLE, CmpGT, CmpGE** : These operations are used to generate code to perform a numeric comparison of the top two operands on the virtual operand stack. The two operands can be real, integer, or boolean types. If the two operands are both constants, the appropriate comparison is performed at compile-time and the resulting constant boolean result is pushed back to the virtual operand stack. If the operands are not both constants, code is generated to perform the appropriate comparison. The resulting value pushed back to the virtual operand stack is a condition code operand. The CondField field for this operand is set according to the type of comparison operation performed. For example, if the CmpEQ operation is used, the CondField field for the resulting...
operand would have a value of CondEQ.

4.4.3.8 File Operations

AssignFile: This operation is used to generate the executable code that performs the file name assignment function. Two operands are removed from the virtual operand stack by this operation. The top operand is the string result which contains the name to be assigned to the file. The second operand is the address of the file or text file variable which is being assigned. This operation generates a call to the operating system which performs the file name assignment.

ResetFile, RewriteFile, CloseFile: These operations are used to generate executable code that opens existing files for reading, creates new files for writing, or closes an open file, respectively. Each of these operations removes one operand from the virtual operand stack. This operand represents the address of the file or text file variable being used. This address is loaded into an address register. A call is then generated to the operating system to perform the appropriate system level task.

SelectStdIO, SelectIO, ReleaseIO: The code generated to output variables is identical for screen and file operations. Prior to any input or output, a SelectStdIO operation or a SelectIO operation is performed. SelectStdIO generates code to push a value of negative one onto the processor return stack. Subsequent IO operations will see this value and send output to the screen. SelectIO removes the top operand from the virtual operand stack. This operand is the address of a
text file variable to be used for IO operations. This address is then pushed onto
the processor return stack. At the completion of an IO operation, the selected
output device or file is removed by issuing a ReleaseIO operation. This operation
simply produces code to remove the screen identifier or text file address from the
processor return stack, leaving the stack as it was prior to the IO function.

**PrintInt, PrintReal, PrintChar, PrintString, PrintEol**: These operations are
used to generate executable code for screen and file output. The generated code
loads the appropriate values into processor registers and generates a call to the
operating system to print out an integer, a real number, a character, a string, or
and end of line sequence. In all instances, the outputted data is sent to the screen
or to a selected file, as specified by a prior SelectStdIO or SelectIO operation.

**ReadVar**: This operation is used to produce code that reads in a value from
the user keyboard or from a text file. The value is then assigned to a real, integer,
char, or string variable. The top operand on the virtual operand stack is the
address of the variable where the read value is to be stored. Param contains a
pointer into the type table indicating the type of the variable being read. The
input is read from the device indicated by the value on top of the processor return
stack: A value of negative one indicates keyboard input. Other values are
interpreted as the address of a text file variable to be used for the input. This
operation generates a call to the operating system to read in the appropriate value.
The operating system checks to see that an appropriate input string is encountered.
For example, it is illegal to read an alpha string into an integer variable. The
operating system then returns stores the appropriate value into the variable address.

**ReadFile, WriteFile** : These operations are used to perform input and output using typed files. The types of the files can be simple types, such as integer, or more complex types such as records or arrays. Two operands are removed from the virtual operand stack. The top operand represents the address of the typed variable to be read or written. The second operand represents the address of the type file variable. Param contains a pointer into the type table which indicates the type of the file variable. Using this data, it is also possible to determine the type of the variable and the size, in bytes, of the data to be stored in the variable. The size of the variable, the address of the variable, and the address of the file variable are all loaded into processor registers. A constant value is loaded into processor data register zero to indicate if a read or a write is to be performed. A TRAP #2 instruction is then used to call the operating system. The operating system performs the file operation and then returns to the calling program.

**ChkEOF** : This operation is used to check for the end of file condition for the file whose address is represented in the top operand on the virtual operand stack. This address is loaded into address register zero. A constant value is loaded into processor data register zero to indicate that a check end of file function is to be performed. A TRAP #2 instruction is then used to call the operating system. The operating system returns a value of one in data register zero if the end of file condition is true or a value of zero if the end of file condition is false. This value
is then represented as a boolean operand and pushed onto the virtual operand stack.

ClrInputLine : This operation is invoked in conjunction with the Pascal readln statement with no input variable parameters specified. The result of such a statement is to skip over the remainder of the current input line from the text file being read, or to ignore the remainder of a user's input line from the keyboard. At an operating system level, this function is performed by clearing the appropriate buffer. The code generated by ClrInputLine performs this function. The device or file for which the input buffer is cleared is indicated by the value on top of the processor return stack: A value of negative one indicates keyboard input. Other values are interpreted as the address of a text file variable being used for input. The generated code loads a constant into data register zero, indicating the function to be performed, and executes a TRAP #2 instruction which calls the operating system.

4.4.3.9 Variable Initialization

InitPF : This operation is invoked at the beginning of each program, procedure, function, and process. Its purpose it to initialize all system variables declared in the current block level to a known state. Process variables are all set to indicate that they are not currently executing. Channels and files are all set to indicate that they are closed. Resource variables are all set to indicate that no process is currently executing within the appropriate resource. Additionally,
address space is allocated for the resource local variables. Shared variables first allocate the necessary global memory space, then set their status fields to indicate that no processes are currently using the shared variable.

ClosePF: This operation is invoked at the end of each program, procedure, function, and process. Similar in concept to the InitPF operation, ClosePF places all system variables of the current block level into a known state. The memory reserved for resource and shared variables is released. Any templates with locks on shared variables perform releases. Any opened files are closed.

4.4.3.10 Shared Variable Operations

ReqRead, ReqWrite: These operations are used to initiate a request for read or write access through a specified template to a specified shared variable. Two operands are removed from the virtual operand stack by these operations. The top operand is the address of the template variable to be granted the rights. The second operand is the address of the shared variable to be accessed using the template. These operations first generate code that inspects the template's current status. If the template current has access to a shared variable, a release operation is first performed. Next, the address of the shared variable is copied into a field of the template variable. Finally, the address of the template variable is pushed onto the processor return stack. This address will again be used for storing the selected range values into the template variable prior to calling the operating system.
**TempRangeStore**: The ReqRead and ReqWrite operations are followed by one TempRangeStore operation for each dimension of the shared variable array type. Param contains the number of the array index for which the range values are being specified. For example, when Param has a value of one, the first array index range is specified, and so on. This operation removes two operands from the virtual operand stack. The top operand is the upper limit on the array index range, and the second operand is the lower limit. The code generated by this operation copies these values into the appropriate fields of the template variable, as determined by the value of Param. As mentioned before, the address of the template variable is found on top of the processor return stack, having been left there by the ReqRead or the ReqWrite operation.

**FinishTemplateOp**: This operation is used after all array index ranges have been stored into a template variable by one or more invocations of the TempRangeStore operation. At this time, the template variable is initialized to indicate the type of access being requested, the address of the shared variable to be accessed, and the ranges of each array index where legal accesses can be made. The FinishTemplateOp operation completes the request function by removing the address of the template variable from the processor return stack and copying it into address register zero, loading a constant value into data register zero to indicate the desired operating system function, and generating a TRAP #2 instruction to call the operation system. The operating will suspend the calling process until the read or write access request is granted.
**TempRelease**: This operation is used to generate code to release any shared variables currently granted to a template variable for read-only or read-write access modes. The TempRelease operation removes one value from the virtual operand stack. This value is the address of the template variable being released. The generated code loads this address into address register zero. Next, a constant value indicating the appropriate operating system function to be performed is loaded into data register zero. Finally, a TRAP #2 instructions is generated. This causes a call to the operating system. The operating system will perform the necessary tasks involved in releasing a template variable's access rights.

**PrepareTemplateWrite, PrepareTemplateRead**: These operations are used each time a template variable is referenced in order to verify that the template variable has been granted sufficient rights to access a shared variable. A template variable is represented as a record type containing several fields. These operations check the Status field of the indicated template variable. The status field can contain three different values: A value of zero indicates that the template is not granted permission to access any shared variable. A value of one indicates that the template is granted read-only access to a shared variable. A value of three indicates that the template is granted read-write access to a shared variable.

The PrepareTemplateWrite operation is performed prior to assigning a value to a shared variable through a template. PrepareTemplateWrite produces code that checks bit one of the template's status field. If this bit is set, the template can proceed to write to the shared variable. If this bit is cleared, the template has not
received write access to the shared variable, and a run-time error is generated.

The PrepareTemplateRead operation is performed prior to reading a value from a shared variable through a template. PrepareTemplateRead produces code that checks bit zero of the template's status field. If this bit is set, the template can proceed to read from the shared variable. If this bit is cleared, the template has not received read access to the shared variable, and a run-time error is generated.

After inspecting the access rights of a template, these operations push the address of the template onto the processor return stack. This address is removed from the virtual operand stack. Next, the base address of the shared variable to be accessed is extracted from the appropriate field of the template variable. This base address is then pushed onto the virtual operand stack. Subsequent array index and record field offset calculations will be added to this base address to determine the final shared memory address to be used.

ChkTempIndex : This operation is used to generate any code necessary to determine if the array index value on top of the virtual operand stack is within the bounds granted to a template variable. The address of the template variable is on top of the processor return stack. Param contains the number of the array index for which the top operand on the virtual operand stack represents. For example, when Param has a value of one, the top operand is the first array index, and so on. Using Param, the compiler determines the appropriate offset into the template variable data structure where the legal range of values are stored for the
designated array index. The top operand on the virtual operand stack is checked to be within the specified range. If it is not, a run-time error is generated. Otherwise, the index value is left on the virtual operand stack so that it can be used to calculate the address of the array element.

**ChkTempIndexRow** : This operation is similar to the ChkTempIndex operation, in that it verifies that an array index value is within the legal bounds granted to the template variable by a read or write access request. The difference is that the ChkTempIndexRow checks to see that the entire row or column of the shared array variable is accessible using the template. This operation is needed since entire rows or columns of arrays can be read or written using a assignment statement. As with the ChkTempIndex operation, Param contains the number of the array index for which the entire row or column must be accessible. For example, when Param has a value of one, the row being referenced is the first array index, and so on. Using Param, the compiler determines the appropriate offset into the template variable data structure where the legal range of values are stored for the designated array index. The base address of the template variable is on top of the processor return stack. The range for the appropriate array index is inspected to see if it corresponds the entire row. If it does not, a run-time error is generated.

**FinishTemplateCheck** : This operation is used after all array index values in a template access are validated. This operation removes the address of the template variable from the top of the processor return stack.
4.5 Syntactic Analysis

As implied by name, the syntactic analyzer determines correctness of the syntax of the source program code. Several other tasks are also performed by this portion of the compiler. These include name determination and assignment, compiler table modification, label declaration, and pseudo-code determination.

The syntactic analyzer scans the source code by obtaining input tokens from the lexical analyzer, and performs appropriate actions based upon the characteristics of the input tokens obtained. In the broadest sense, the syntactic analyzer performs two types of actions: 1) store values into the compiler tables, and 2) perform calls to the code generator to produce executable machine language instructions.

For example, when the input tokens indicate that the user program is declaring a constant, the syntactic analyzer obtains all tokens contained in the constant declaration and stores the appropriate values in the compiler tables. When the constant identifier is later encountered in the user program, inspection of these tables will indicate that 1) the identifier represents a constant value, and 2) what the value of that particular constant is.

When scanning an executable statement, such as an assignment statement, the syntactic analyzer first scans in the name of the variable receiving the assignment. The compiler tables are then inspected to verify that the identifier represents a variable name. When this is verified, the syntactic analyzer performs
a call to the code generator, causing it to generate code which will calculate the
memory address of the specified variable. Next, the syntactic analyzer performs
a series of token fetches and code generation calls in order to evaluate the value
to be assigned. Finally, a call is made to the code generator to perform the actual
storage, completing the assignment statement.

The procedures used by the syntactic analyzer can be broadly divided into
two classes: housekeeping procedures and parsing procedures. Each of these
classes are discussed in the following sections.

4.5.1 Syntactic Analysis Housekeeping Procedures

The procedures discussed here are used to perform repetitive tasks in a
unified manner. The first procedure to be discussed is FlagError. The declaration
is as follows:

```
procedure FlagError(ErrorNumber:integer);
begin
  writeln('Error in Compilation');
  writeln(ErrorMessage(ErrorNumber));
  writeln('Error Number : ',ErrorNumber);
  writeln('Approximate Error Location :');
  writeln(' Line : ',CurrentLineNumber);
  writeln(' Column : ',CurrentColumnNumber);
  writeln(' File : ',CurrentFileName);
halt;
end;
```

This procedure is called whenever a compilation error occurs. For example, if the
compiler is at a point in the source program where it expects to find the reserved
and some other input token is encountered, the compiler will call the FlagError procedure. The integer argument used in this case would be the error number indicating that the reserved word begin is expected. The function ErrorMessage referenced in the above procedure returns a string value that corresponds to the error number passed as an argument. For the error number indicating that the reserved word begin is expected, the function ErrorMessage would return the string "Begin Expected."

The FlagError procedure proceeds to report where the error is located. This is done using the global variables CurrentLineNumber, CurrentColumnNumber, and CurrentFileName. These variables indicate the line number, column number, and file name, respectively, where the error occurred.

When compiling Pascal, it is very common to merely check for the existence of a reserved word input token. For example, when parsing an if statement, the reserved word then is expected immediately following the boolean expression. Nothing is done with the reserved word. Because this type of structure occurs frequently in Pascal, a special procedure named Expect is used. It is declared as follows:

```pascal
procedure Expect(ExpectedInputToken:TokenSymType; 
                   ErrorNumber:integer);
begin
  if LexToken.Sym <> ExpectedInputToken then
    FlagError(ErrorNumber);
  Get;
end;
```
As can be seen, the procedure Expect checks to see that the correct input token is present. Once this is determined, the next input token is obtained.

The next procedure to be discussed is AdjustAddr. This procedure is used to properly align word and long word addresses and offsets, as described above. The procedure is as follows:

```
procedure AdjustAddr( var AddressValue : longint; Mask : longint);
begin
  AddressValue := (AddressValue + Mask) AND (not Mask);
end;
```

It should be noted that the parameter AddressValue is a var parameter; the address of the variable is passed to the procedure so that the variable’s value is changed in the calling procedure. Also, the parameter is of type longint. Therefore, any addresses to be modified using this procedure must be stored in longint variables.

The function NewLabel is used frequently throughout the compilation of a program. This function returns a unique integer value for use as a label number within the executable code. Labels are used in a variety of different situations within BPL programs. For example, each procedure, function, or process is executed by jumping to the label assigned to it. Additionally, conditional and iterative statements use labels for branching within a statement.

The intermediate language operation DecLabel is used to assign an address within the executable code to a label number. The operations JmpTrue, JmpFalse, and JmpAlways are used to generate the appropriate branch instructions to a
specific label number.

Two tables are maintained internally by the compiler for resolving label references, the Label Table and the Unresolved Reference Table. The Label Table contains the label numbers and corresponding addresses for all labels that have been assigned using the DecLabel operation. The Unresolved Reference Table contains the label numbers for all "forward" referenced labels. This situation occurs when a jump is made to a higher code address. In the Unresolved Reference Table, the address of a forward jump instruction is stored along with the label number for each forward reference.

These two tables are maintained in the following manner. When a jump instruction is coded, the Label Table is searched to see if the jump label is defined. If it is already defined, the jump instruction can be completely specified. If the label is not defined, the address of the jump instruction and the label number are added to the Unresolved Reference Table.

When the intermediate operation DecLabel is performed, the label number and the current code address are added to the Label Table. Next, the Unresolved Reference Table is searched to see if any jumps have been coded to this label number. If any are found, the address of the jump instruction is used to insert the correct code. These references are then considered to be resolved, and are therefore removed from the Unresolved Reference Table.

As mentioned above, jump instructions and labels are used extensively in the translation of BPL programs. Because of this, the Label Table could become
quite large. This problem is prevented by selectively removing entries from the Label Table using the ForgetLabel procedure.

Many labels correspond to jump destinations created in order to translate such conditional and iterative BPL statements such as if then else statements, for loop statements, while do loop statements, case statements, and repeat until loop statements. The labels used within these statements are not used outside the statement. Therefore, after parsing the above types of statements, the parsing routines call the ForgetLabel procedure to remove each control statement label from the Label Table. As a result, at the end of the compilation of a program, the only labels remaining in the Label Table are those labels corresponding to procedures, functions, and processes. The number of these types of labels is relatively small, thus facilitating the use of a small Label Table.

4.5.2 Syntactic Analysis Parsing Procedures

The parsing portion of the BPL compiler consists of a set of recursive descent procedures, as described in [Aho,1977], [Davie,1981], and [Hartman,1977]. The basic technique used to construct the compiler is as follows:

1. Create the BNF representation of the language specification.

2. For each non-terminal defined in the language specification, create a procedure to parse the non-terminal.

The BNF representation of the language specification consists of equations defining non-terminals, as described in Chapter III. Each non-terminal is defined
using terminals and non-terminals. It is possible for a given non-terminal to appear as component of its own definition. Additionally, a given non-terminal $A$ can be a component of another non-terminal $B$ definition while the non-terminal $B$ is a component of the definition of non-terminal $A$. These situations are responsible for the recursive nature of the parsing procedures.

The following sections will present simplified versions of many of the recursive descent parsing procedures. Each will be presented along with the BNF representations of the applicable non-terminals.

### 4.5.2.1 Main Program Parsing

The BNF definition of a BPL program is as follows:

```
<BPL program> == [ <program heading> ] <body> .
<program heading> == program <identifier> ;
```

In other words, a BPL program consists of an optional program heading, a body, and a terminal period (.)

The procedure for parsing this non-terminal would therefore first check to see if a program heading is present. Because a program heading must begin with the terminal `program`, this check can be performed merely by inspecting the input symbol to see if it is this terminal. If it is, the program heading should be parsed. Next, the procedure would call another procedure to scan the non-terminal, `<body>`. Finally, the procedure would inspect the input symbol to verify that a period follows the main body. The
pseudo code for this procedure, as well as the procedure for parsing the program heading, is as follows:

```pascal
procedure ScanProgram;
begin
  { Check for a program heading. }
  if LexToken.Sym = program1 then
    ScanProgramHeading;
  { Scan the main body of the program. }
  ScanBody;
  { Check for a period following the body. }
  Expect(period1,23);
end;

procedure ScanProgramHeading;
begin
  { Get rid of the program1 token. }
  Get;
  { Check for the presence of an identifier. }
  Expect(identifier1,2);
  { Check for the required semicolon. }
  Expect(semicolon1,14);
end;
```

Two points need to be made regarding these procedures. Firstly, the error numbers 23, 2, and 14 correspond to error messages "Expected", "Identifier Expected", and "Expected", respectively. Also, it should be noted that the program heading is not processed in any way other than to determine that the terminal `program` is followed by an identifier and a terminal `. ` Nothing is done with the name of the program.

The BNF representation of the non-terminal `<body>` is as follows:
\[ <\text{body}> = = <\text{declarations}> <\text{compound statement}> \]

\[ <\text{declarations}> = = [ <\text{declaration}> ]^* \]

\[ <\text{declaration}> = = <\text{const declaration}> | <\text{type declaration}> | <\text{function declaration}> | <\text{procedure declaration}> | <\text{variable declaration}> \]

Therefore, the procedure to scan a body will be very simple. It will merely call two other procedures. The first procedure called will be the procedure to scan declarations, while the second is the procedure to scan a compound statement.

\[
\text{procedure ScanBody;}
\begin{align*}
\text{begin} \\
\{ \text{ Scan the declarations. } \} \\
\text{ScanDeclarations;} \\
\{ \text{ Scan the compound statement. } \} \\
\text{ScanCompoundStatement;} \\
\text{end;}
\end{align*}
\]

4.5.2.2 Declarations Parsing

The procedure to scan declarations is slightly more complex, due to the complexity of Pascal declarations. The declarations section may contain constant declarations, type declarations, variable declarations, function declarations, or procedure declarations. The different variations of declarations may occur in any order and in any number. It is also legal for the declarations section to contain no declarations.

Fortunately, each of the different types of declarations can easily be recognized by the first symbol of the declaration. The reserved words \texttt{const}, \texttt{type},
var, function, and procedure indicate the beginning of a constant, type, variable, function, or procedure declaration, respectively. Using these reserved words to determine the action of the compiler allows the following implementation of the ScanDeclarations procedure:

```pascal
procedure ScanDeclarations;

var
  DeclarationsDone : boolean;
begin
  { Initialize DeclarationsDone to false. }
  DeclarationsDone := false;
  { Call the appropriate procedure depending upon the next input symbol.
    Do this repeatedly until there are no more declarations to scan. }
  while not DeclarationsDone do
    case LexToken.Sym of
      { If the next input symbol is the reserved word "const," scan
        a constant decl. }
      const1 : ScanConstDeclaration;
      { If the next input symbol is the reserved word "type," scan
        a type declaration. }
      type1 : ScanTypeDeclaration;
      { If the next input symbol is the reserved word "var," scan a
        variable decl. }
      var1 : ScanVarDeclaration;
      { If the next input symbol is the rsvd word "procedure," scan
        a procedure decl. }
      procedure1 : ScanProcedureDeclaration;
      { If the next input symbol is the reserved word "function,"
        scan a function decl. }
      function1 : ScanFunctionDeclaration;
      else 
        { If the next input symbol is none of the above reserved
          words, there are no more declarations. }
      end
    DeclarationsDone := true;
  end;
end;
```

Note that, if there are no declarations, the above procedure will fall through
without calling any of the declaration parsing routines. Additionally, no restrictions are imposed on the ordering of the various types of declarations nor on the number of declaration types.

4.5.2.2.1 Constant Declaration Parsing

The procedure ScanConstantDeclarations is the first procedure discussed that performs any action other than calling other procedures. The BNF definition of a constant declaration section is as follows:

\[
<\text{const declaration}> = \text{const} <\text{constant definition}> ; [ <\text{constant definition}> ; ]^* \\
<\text{constant definition}> = <\text{identifier}> = <\text{constant}> \\
<\text{constant}> = <\text{integer constant}> \mid <\text{boolean constant}> \mid <\text{string constant}> \mid <\text{char constant}> \mid <\text{real constant}>
\]

In addition to determining the syntactic accuracy of constant declarations, the procedure ScanConstantDeclarations will also call a procedure to store information for each identifier declared as a constant. As described in section 4.3.2 Compiler Table Procedures, the procedure DecConst is used to perform this task.
procedure ScanConstantDeclarations;
var
  IdentifierName : integer;
begin
  { Get rid of the reserved word const. }
  Get;
  { Scan constant def's until there are no more. }
  while LexToken.Sym = identifier1 do begin
    { Save the name of the identifier. }
    IdentifierName := LexToken.IntVal;
    { Get rid of the identifier. }
    Get;
    { Make certain an "=" is next. }
    Expect(equals1,16);
    { See what type of constant is in the input stream; store the data accordingly. }
    case LexToken.Sym of
      intconstant1 : DecConst(IdentifierName, IntConst, LexToken.IntVal);
      charconstant1 : DecConst(IdentifierName, CharConst, LexToken.IntVal);
      stringconstant1 : DecConst(IdentifierName, StrConst, LexToken.IntVal);
      realconstant1 : DecConst(IdentifierName, RealConst, LexToken.IntVal);
      true1 : DecConst(IdentifierName, BoolConst, 1);
      false1 : DecConst(IdentifierName, BoolConst, 0);
      else FlagError(50);
    end;
    { Get rid of the current input symbol. }
    Get;
    { Verify the presence of a semicolon }
    Expect(semicolon1,14);
  end;
end;
end;

Note that, if the input symbol following the equals sign is not one of the symbols specified above, the FlagError procedure is called with a parameter of 50. This value corresponds to the error message "Error in Constant." The compiler will print several lines of the source text prior to and including the line where the error
occurred. It will then print out the error message and halt.

4.5.2.2.2 Type Declaration Parsing

The procedure to scan type declarations is similar to that for scanning constant declarations in that it calls a procedure to declare an identifier as being a type name. The BNF specification is as follows:

\[
\text{<type declaration>} \equiv \text{type} \ <\text{type definition}> ; \ [ \ <\text{type definition}> ; ]^* \\
\text{<type definition>} \equiv \ <\text{identifier}> \equiv \ <\text{type}> \\
\]

A procedure named ScanType is called to parse the non-terminal \(<\text{type}>\). This procedure is passed a single integer var parameter. The procedure returns in this parameter the index number of the element of the type table where the data corresponding to the scanned type is stored. The procedure for scanning type declarations is as follows:
procedure ScanTypeDeclarations;
var
  IdentifierName : integer;
begin
  { Get rid of the reserved word type. }
  Get;
  { Scan type def's until there are no more. }
  while LexToken.Sym = identifier1 do begin
    { Save the name of the identifier. }
    IdentifierName := LexToken.IntVal;
    { Get rid of the identifier. }
    Get;
    { Make certain an "=" is next. }
    Expect(equals1,16);
    { Call the procedure that scans the type. }
    ScanType;
    { Declare that the identifier is a type name. }
    DecType(IdentifierName, TypeTab1Ptr);
    { Verify the presence of a semicolon }
    Expect(semicolon1,14);
  end;
end;
end;

The procedure ScanType performs a complex function, largely due to the large number of data types possible in BPL.

<type> == <simple type spec> | <record type spec> | <array type spec> | <file type spec> | <subrange type spec> | <channel type spec> | <template type spec> | <shared type spec> | <user declared type spec> | <resource type spec> | <process type spec>

<simple type spec> == longint | integer | word | shortint | byte | boolean | char | string | text

<record type spec> == record <field list> : <type> ; [ <field list> : <type> ; ]* end

<field list> == <identifier> [ , <identifier> ]*

<array type spec> == array [ <type> [ , <type> ]* ] of <type>
<file type spec> == file of <type>

<subrange type spec> == <constant value> .. <constant value>

<channel type spec> == channel of <type>

<template type spec> == template of <type>

<shared type spec> == shared <type>

<user declared type spec> == <identifier>

The BNF descriptions for process and resource type specifications are presented in sections 3.3.1 The Process and 3.1.2 The Resource, respectively.

As with the different classes of declarations, each different class of type description can be identified by the first symbol of the description. For example, all record type descriptions begin with the reserved word record while all array type descriptions begin with the reserved word array. Therefore, the ScanType procedure will investigate the current input symbol and, depending upon this symbol, call an appropriate procedure to parse the type description:
procedure ScanType(var TypePtr:integer);
begin
  case LexToken.Sym of
    longint1, integer1, word1, byte1, shortint1, boolean1, string1, char1,
    text1 : ScanSimpleType(TypePtr);
    file1 : ScanFileType(TypePtr);
    record1 : ScanRecordType(TypePtr);
    array1 : ScanArrayType(TypePtr);
    intconst1, boolconst1, charconst1 : ScanSubrangeType(TypePtr);
    channel1 : ScanChannelType(TypePtr);
    template1 : ScanTemplateType(TypePtr);
    shared1 : ScanSharedType(TypePtr);
    identifier1 : ScanUserDeclaredType(TypePtr);
    resource1 : ScanResourceType(TypePtr);
    process1 : ScanProcessType(TypePtr);
    else FlagError(10);
  end;
end;
end;

As can be seen, the ScanType procedure calls one of several procedures to scan a type specification depending upon the value of the current input symbol. Note also that, if the current input symbol does not correspond to any of the legal types, the compiler will call the FlagError procedure with an error number argument of ten. This error number corresponds to the error message "Error in Type."

The procedure ScanType and all of the procedures called by ScanType for scanning the different classes of type descriptions are declared with a single var integer parameter. Each of the procedures will return in this parameter the index number of the element within the type table that contains the data describing the type. The procedure ScanType passes its parameter each procedure it calls in order to scan a type description. Therefore, ScanType always returns the index
The procedure ScanSimpleType is used to scan the identifier of one of the predefined types. For each of these predefined types, an entry is placed into the type table during compiler initialization. Additionally, an integer variable associated with each simple type indicates the position within the type table of the description of the simple type. For example, the integer variable LongintTypePtr contains the index number of the element within the type table of the description of the simple type longint. The procedure for scanning simple types appears below:

```
procedure ScanSimpleType(var TypePtr:integer);
begin
  { Store the appropriate element pointer value into TypePtr depending upon the input symbol. }
  case LexToken.Sym of
  longintl : TypePtr := LongintTypePtr;
  integerl : TypePtr := IntegerTypePtr;
  wordl : TypePtr := WordTypePtr;
  bytel : TypePtr := ByteTypePtr;
  shortintl : TypePtr := ShortintTypePtr;
  booleanl : TypePtr := BooleanTypePtr;
  stringl : TypePtr := StringTypePtr;
  charl : TypePtr := CharTypePtr;
  textl : TypePtr := TextTypePtr;
  end;
  { Get rid of the current input symbol. }
  Get;
end;
```

The procedure ScanFileType is used for scanning type descriptions of type files. Unlike the ScanSimpleType procedure, where the simple type descriptions are already stored in the type table, the ScanFileType procedure will scan a new
type which therefore must be added to the type table. As described in section 3.2.1 Compiler Table Structure, the data added to the type table includes an indicator that the type is a file type, the element within the type table of the description of the file's component type, the size of the file type, and the block level of the type. The procedure for parsing file type specifications is as shown below:

```plaintext
procedure ScanFileType(var FTypePtr : integer);
var
  CompType : integer;
  { pointer to the description of the component type}
begin
  { Get rid of the reserved word FILE. }
  Get;
  { Check for the reserved word OF. }
  Expect(ofl,8);
  { Scan the description of the component type.}
  ScanType(CompType);
  { Store the type description in the type table. }
  StoreTypeData(FileType,CompType,0,0,90);
end;
```

The ScanFileType procedure is recursive in nature due to the fact that it calls the ScanType procedure in order to determine the type of the file component. In turn, the ScanType procedure can call any of the procedures to scan types, including ScanFileType.

The ScanRecordType procedure is used to parse declarations of record data types. This procedure is also recursive in nature since it calls the ScanType procedure in order to parse the type of each record field. The ScanType procedure could then call the ScanRecordType procedure if, for example, the type of one of
a record's fields is a record type.

```pascal
procedure ScanRecordType(var RecTypePtr : integer);
var
  LocalFieldTable : array [1..64] of FieldDesc;
  TotalCount, {total number of fields}
  ThisFieldListCount, {number of fields per type}
  i : word;
  FieldTypePtr : integer;
  RecTypeMask,
  FieldOffset : longint;
begin
  Get; {RECORD}
  TotalCount := 0; {no fields yet}
  FieldOffset := 0; {no fields = no offset yet}
  RecTypeMask := 0;
  while LexToken.Sym = identifier1 do begin
    ThisFieldListCount := 1; {use this to count}
    LocalFields[TotalCount+ThisFieldListCount].FName := LexToken.IntVal;
    Get; {Get rid of the identifier}
    while LexToken.Sym = comma1 do begin
      Get; {Get rid of the comma}
      if LexToken.Sym <> identifier1 then
        FlagError(2);
      inc(ThisFieldListCount); {count the identifier}
      { and save its name }  
      LocalFields[TotalCount+ThisFieldListCount].FName :=
        LexToken.IntVal;
      Get; {Get rid of the identifier}
    end;
  {At this point, there are no more commas. Therefore, there are no more
  identifiers either. We should now find a colon followed by a
  type specification followed by a semicolon.}
    Expect(colon1,5);
    ScanType(FieldTypePtr);
    Expect(semicolon1,14);
  { Now we have the names of all identifiers in this field list and a pointer
  to the type of the fields. Next, we must store, in our local
  field table, the type of each of these fields and the address
  offset of each field. While doing so, we must also tabulate
  the total size of the record type, the word alignment mask for
  each field, and the word alignment mask for the entire record
  type. }
```
for i := 1 to ThisFieldListCount do
  with LocalFields[TotalCount + i] do begin
    FTypePtr := FieldTypePtr;
    AdjustAddr(FieldOffset, TypeTabl^ [FieldTypePtr]. AddrMask);
    AddrOffs := FieldOffset;
    FieldOffset := FieldOffset + TypeTabl^ [FieldTypePtr]. TypeSize;
    RecTypeMask := RecTypeMask OR TypeTabl^ [FieldTypePtr]. AddrMask;
  end;
  TotalCount := TotalCount + ThisFieldListCount;
end;
{ Make certain the END is present. }
Expect(endl,13);
{ At this point, the entire record type is parsed. No data regarding the type, however, is yet stored in the type table or the field table. First, we store the type definition in the type table, using the current value of FieldTablPtr as the index of the first field description, TotalCount as the total number of fields, the final offset value as the size of the record type, and the logical OR'ing of all the field type's address masks as the record type's masks.}
StoreTypeData(RecordType, 0, FieldTablPtr, TotalCount, FieldOffset, RecTypeMask);
{ Now, store the data for each field into the field table, and update the FieldTablPtr to reflect the new total number of user-specified fields.}
for i := 0 to TotalCount - 1 do
  FieldTabl^ [FieldTablPtr + i] := LocalFields[i+1];
  FieldTablPtr := FieldTablPtr + TotalCount;
{ Finally, copy the current value of TypeTablPtr into the var parameter RecTypePtr. Since we just used the StoreTypeData procedure to store data describing the record type, TypeTablPtr is the index number of the element in the type table where the record type is described. }
  RecTypePtr := TypeTablPtr;
end;

Several points regarding this length procedure will be discussed. First of all, note that the procedure maintains its own list of all field names, types, and address offsets, rather than store each directly in the field table. The reason for this is that the field table contains a contiguous list of all fields in a record type. The fields
for a record type must be stored in the field table consecutively and with no other fields in between. During the scanning of the field types, however, it is legal for a field type to also be a record type. If the field data were stored while each field is parsed, the fields of any field record type would be stored in the field table, disrupting the sequential listing of the fields for the "outer" record type. Therefore, the ScanRecordType procedure maintains its own local field table in which data for all fields within the record is stored. At the completion of the ScanRecordType procedure, the data for these fields is copied into the field table, and the variable FieldTablPtr is updated to reflect the new entries.

The variable FieldOffset is updated throughout the procedure, such that it always contains the address of the first byte following the last field of the record type. Therefore, the next field can be stored at the offset stored in FieldOffset. Before using this address, however, it is adjusted for proper word alignment by calling the AdjustAddr procedure. The bit mask used to perform this adjustment is the address bit mask of the field type. Therefore, the offset of the field within the record type will be properly aligned. After each field's address offset is stored, the variable FieldOffset is incremented by an amount equal to the size of the field variable's data type. The next field, therefore, will be stored immediately following the previous. After the parsing of the last field, the variable FieldOffset points to the first free byte after the last field's data storage area. This value is equivalent to the total size of the record type, and is used as such.

While parsing record types the logical ORing of all field address bit masks
is maintained in the variable RecTypeMask. This yields the overall address bit mask for the record type. When the base memory address of a variable of the record type is word-aligned in accordance with the value of the address bit mask for the record type, each field component of the record type will also be properly aligned.

Array type declarations are parsed using the procedure ScanArrayType. A simplified version of this procedure is presented below. The simplification is such that the procedure will only parse single dimensional arrays.
procedure ScanArrayType(var ArrayTypePtr : integer);
var
    IndexTypePtr,
    ElementTypePtr : integer;
    ArrayStride,
    ArraySize,
    ArrayAddressMask : longint;
begin
    Get; {ARRAY}
    Expect(leftbracket1,11);
    { Parse the index type. }
    ScanType(IndexTypePtr);
    Expect(rightbracket1,12);
    Expect(ofl,8);
    { Parse the element type. }
    ScanType(ElementTypePtr);
    { Calculate the spacing between elements of the array}
    ArrayStride := TypeTabl^[ElementType].TypeSize;
    AdjustAddr(ArrayStride, TypeTabl^[ElementType].AddrMask);
    { Calculate the size of the array type. }
    ArraySize := (TypeTabl^[IndexType].P2 - TypeTabl^[IndexType].P1 +
                  1) * ArrayStride;
    { Get, from the element type, the type address bit mask for the array type.}
    ArrayAddressMask := TypeTabl^[ElementType].AddrMask;
    { Store the data in the type table. }
    StoreTypeData(ArrayType, 0, IndexTypePtr, ElementTypePtr, ArraySize,
                  ArrayAddressMask);
    { Return the pointer to the type description in the parameter. }
    ArrayTypePtr := TypeTablPtr;
end;

The ScanArrayType procedure uses two integer variables, IndexTypePtr and
ElementTypePtr, to point to the data in the type table corresponding to the index
type and the element type, respectively. The index type specifies the legal range
of index expression values. The index type also specifies the number of elements
in the array. The overall size of the array is determined by multiplying the
number of elements in the array by the array stride. The array stride is the
spacing between adjacent elements of the array. Without word alignment correction, the stride would be equal to the size of the array element type. Each element of the array, however, must be properly word-aligned in memory. Therefore, the array stride is first set equal to the element type size. That value is then adjusted using the procedure AdjustAddr. Finally, the address bit mask for the array type is equivalent to the address bit mask of the element type.

Subrange types are similar to simple longint, char, and boolean types. The distinction is that subrange types allow only a limited, specified group of values within the simple type. This group is designated by an upper and lower bound, separated by the terminal consisting of two periods (..). The procedure for scanning subrange types is shown below:
procedure ScanSubrangeType(var SubTypePtr : integer);
var
  ExpectedType : longint;
  LowerBound,
  UpperBound : longint;
begin
  { Store the lower bound. }
  LowerBound := LexToken.IntVal;
  { Determine if this is int, char, or bool. }
  case LexToken.Sym of
    intconst1 : ExpectedType := LongintTypePtr;
    boolconst1 : ExpectedType := BooleanTypePtr;
    charconst1 : ExpectedType := CharTypePtr;
  end;
  { Advance ahead to the next input symbol. }
  Get;
  { Check for the .. }
  Expect(DotDot1,22);
  { Verify type compatibility. }
  case LexToken.Sym of
    intconst1 : if ExpectedType <> LongintTypePtr then FlagError(107);
    boolconst1 : if ExpectedType <> BooleanTypePtr then FlagError(107);
    charconst1 : if ExpectedType <> CharTypePtr then FlagError(107);
  end;
  { Get the upper bound. }
  UpperBound := LexToken.IntVal;
  { Store the type data in the type table. }
  StoreTypeData(SubrangeType, ExpectedType, LowerBound, UpperBound,
    TypeTabl^ [ExpectedType].TypeSize,
    TypeTabl^ [ExpectedType].AddrMask);
  { Return the index of the type data. }
  SubTypePtr := TypeTablPtr;
end;

The procedures for scanning template, channel, and shared types are all simple in nature and identical in form to the procedure for scanning file types. They will therefore not be described here.

User declared types are represented by an identifier which is previously defined as a type in a type declaration. The procedure for parsing such a type
verifies that the identifier corresponds to a valid type declaration. The pointer into the type table of the user declared type is extracted from the identifier type by using the GetIdDesc procedure described in section 4.3.2 Compiler Table Procedures. This value is returned in the integer var parameter.

```
procedure ScanUserDeclaredType(var UserTypePtr:integer);
var
  IdDescription : IdRec;
  IdFound : boolean;
begin
  { Get data from the identifier table. }
  GetIdDesc(LexToken.IntVal, IdDescription, IdFound);
  { Make sure the identifier is declared. }
  if not IdFound then FlagError(104);
  { Make sure the identifier is a type name. }
  if IdDescription.IdTyp <> TypeName then FlagError(103);
  { Store the pointer to the type data. }
  UserTypePtr := IdDescription.ValPtr;
end;
```

Unlike the standard types discussed above, the resource type involves executable statements. In fact, the structure of the ScanResourceType procedure resembles the structure of the procedure parsing routine. The ScanResourceType procedure first checks for the reserved word interface. Next the procedure creates a new resource type entry in the type table. Initially, the new resource type is designated as having zero resource procedures. The procedures StartNewBlock and StartProcessLevel are executed. StartNewBlock is used to create a new block level for the resource, while StartProcessLevel sets up status variables so that the resource will not be able to access variables, procedures, or functions external to the resource.
After entering the new block level, the ScanResourceType procedure scans in the heading of each resource procedure. For each resource procedure heading, the procedure identifier is inspected to make certain it is unique. The identifier is then added to the resource procedure table as an "undefined" resource procedure. Next, the resource procedure parameter declaration, if any, is parsed to determine the name and type of each parameter. The data describing each resource procedure is added to the field table. This data includes the procedure name, the code label of the procedure’s executable code, and the location within the parameter table of the data describing the procedures parameters.

The parser next expects to encounter the reserved word implementation. Following this, the ScanDeclarations procedure is used to parse all local resource constants, types, variables, functions, and procedures. Included within the procedures parsed by the ScanDeclarations procedure are the resource procedures. These procedures are parsed using the parameters declared in the interface portion of the resource declaration. Additionally, when resource procedures are parsed, they are updated within the resource procedure table as "defined." Finally, the procedures EndProcessLevel and EndBlock are called to terminate the resource. Additionally, the EndBlock procedure searches the resource procedure table for any undefined resource procedures. If any are found, an error is reported. In this way the compiler verifies that all resource procedure headings encountered in the interface portion of the resource specification have a corresponding procedure in the implementation portion.
procedure ScanResourceType(var TPtr:integer);
var Row, Col,
    FNum, IProcL : integer;
    FirstProc, NumProc,
    FirstParm, NumParm,
    i, ParmSize : longint;
begin
    Get; {RESOURCE}
    { The reserved word INTERFACE should appear next. }
    Expect(interface1,62);
    { Resource procedures will be stored in the Field Table. Mark the position of the first one. }
    FirstProc := FieldTablPtr + 1;
    { Store the data in the type table. }
    StoreTypeData( ResourceType, DefaultStack, FirstProc, 0, 4, 1);
    { Note the value of the Type Table Pointer. }
    TPtr := TypeTablPtr;
    { Begin a new block level. Since this is a resource, a new process level must also begin. This prevents the resource from accessing global variables. }
    StartNewBlock;
    StartProcessLevel;
    { Scan in the heading of each resource procedure. }
    while LexSym.token=procedure1 do begin
        Get; {PROCEDURE}
        GetStatus(row,col,fnum);
        { Assign a unique label to the resource proc. }
        IProcL := NewLabel;
        { Make certain the proc name is unique. }
        if LexSym.token <> identifier1 then FlagError(2);
        for i := FieldTablPtr downto FirstProc do
            if FieldTabl^[i].FName = LexSym.i1 then
                FlagError(101);
        { Add the resource procedure to the field table. }
        inc(FieldTablPtr);
        if FieldTablPtr > MaxNumFields then FlagError(264);
        DecRProc(LexSym.i1,FieldTablPtr);
        FieldTabl^ [FieldTablPtr].FName := LexSym.i1;
    { Enter a new block level to scan parameters. }
    StartNewBlock;
    FirstParm := ParamTablPtr + 1;
    Get;
    { Scan the resource proc's parameters. }
    if LexSym.token=LeftPar1 then
ScanParmDec(ParmSize,true);
EndBlock;
{ Update the field table with parameter info. }
NumParm := ParamTablPtr - FirstParm + 1;
FieldTabl^[FieldTablPtr].AddrOffs := IProcl;
FieldTabl^[FieldTablPtr].FTypePtr := FirstParm;
FieldTabl^[FieldTablPtr].FTypePcnt := NumParm;
FieldTabl^[FieldTablPtr].Row := Row;
FieldTabl^[FieldTablPtr].Col := Col;
FieldTabl^[FieldTablPtr].FNum := FNum;
{ Semicolon must follow. }
Expect(semicolon1,14);
end;
{ Determine the number of resource procedures 
  and store the result into the type table. }
NumProc := FieldTablPtr - FirstProc + 1;
TypeTabl^[TPtr].p2 := NumProc;
Expect(implementation1,63);
{ Scan all procedures, local procedures, and local 
  variables contained within the resource. }
ScanDeclarations;
{ The reserved word END must follow. }
Expect(end1,13);
{ Determine the size of resource variables 
  and store the result into the type table. }
TypeTabl^[TPtr].VariableSpace := VarAddrs[BlockLevel];
{ Terminate the process level and indicate the end of 
  the block level. The procedure EndBlock will flag 
  an error if it encounters any resource procedures 
  that were not defined. }
EndProcessLevel;
EndBlock;
end;

The parsing of a process is similar to that of a resource. A process contains a single entry point, unlike the multiple entry procedures of a resource. The procedure ScanProcessType begins by adding the process type to the type table. The procedures StartNewBlock and StartProcessLevel are executed to begin a new block level and to prevent the process from accessing global variables, procedures,
and functions. Next, the process parameter declaration is parsed and the results are added to the parameter table.

The next step is to calculate the size of the process variable. Twelve bytes of overhead are required for each process variable. Additionally, extra space is required for each process parameter. For each var and ret parameter, an extra four bytes is required. For structured value parameters, such as records or arrays, an extra four bytes is required. For simple value parameters, such as integers or strings, the size of the process variable in incremented by the size of the type.

The ScanProcessType procedure calculates the size of a process variable by going through the list of parameters and incrementing the size according to the type and size of each parameter. After this size is calculated, the value is written into the correct entry in the type table. The procedure next parses any local constants, types, variables, functions, or procedures by calling the ScanDeclarations procedure.

The next step is to declare the starting address of the process executable code. This is done by invoking the intermediate code operation DecLabel. A new unique label number is assigned to the variable StartLabel, and an unconditional jump to this label is immediately coded. Therefore, the first instruction of any process is a jump instruction. The reason for this is that the BPL compiler stores data in the instruction code space describing the process memory requirements and the characteristics of each process parameter. This data is located immediately after the jump instruction found at the beginning of each process. Since the jump
instruction requires six bytes, the data describing the process memory requirements and parameter types can be found within the program code at the address six bytes following the starting address of the process.

The first data stored after the jump instruction is a long word containing the sum of the stack size and the local process variable size. Following this is a word containing the number of process parameters.

Next, for each process parameter a word and a long word is stored. The low byte of the word contains a one for value parameters, a two for var parameters, and a three for ret parameters. The high byte of the word contains different values for different types of parallel entity parameters, such as channels, shared variables, or resources. The long word contains the total size of the parameter.

The label number stored in the variable StartLabel is now declared using the DecLabel intermediate code operation. This label is destination of the first jump instruction of the process code. The procedure ScanCmpdStat is next called to parse the main body of the process. The parsing of the process is completed by generating the code to terminate a process and terminating the current block level.
procedure ScanProcessType(var TPtr:integer);
var
  ProcessLabel, ParmSize,
  FirstParm, NumParm, StartLabel, i, TypSize, PTypSize: longint;
  MemSize: longint;
begin
  Get;  {PROCESS}
  ProcessLabel := NewLabel;  { Label number for the code }
  FirstParm := ParamTablPtr + 1;
  { Note where the first parameter will be stored. }
  StoreTypeData(ProcessType, ProcessLabel, FirstParm, 0, 12, 3);
  TPtr := TypeTablPtr;
  { Start a new block level. Since this is a process, start a new process
  level to prevent the process from accessing global variables,
  functions, or procedures. }
  StartNewBlock;
  StartProcessLevel;
  { Parse the parameter declaration, if present. }
  if LexSym.Token = LeftParl then ScanParmDec(ParmSize, true);
  { Determine the number of parameters. Store result in the type table. }
  NumParm := ParamTablPtr - FirstParm + 1;
  TypeTabl^[TPtr].p2 := NumParm;
  { Now calculate the size of the process variable:
  1 word for status,
  long word for link address of next process variable,
  long word for code address,
  1 word for cpu number, process number
  }
  PTypSize := 12;
  { add in appropriate size for each parameter }
  for i := FirstParm + NumParm - 1 downto FirstParm do begin
    case ParamTabl^[i].IsVar of
      ValPtype :
        if TypeTabl^[ParamTabl^[i].TypePointer].TypeDesc in
          [RecordType, ArrayType, ChannelType, SharedType, ResourceType]
        then inc(PTypSize, 4)
        else PTypSize := PTypSize + TypeTabl^[ParamTabl^[i].TypePointer].TypeSize;
      VarPtype,
      RetPtype : inc(PTypSize, 4);
    end;
  end;
  PTypSize := (PTypSize + 1) and $fffffff;
end;
  { Update the process type size in the type table. }
TypeTable^[TPtr].TypeSize := PTypSize;
Expect(semicolon,14);
  { Scan any local process declarations, including types, constants,
    variables, procedures, and functions. }
ScanDeclarations;
if LexSym.Token <> begin then FlagError(17);
  Emit(Initialize,0);
  { Place the label for the start of the process. }
Emit(declabel,ProcessLabel);
StartLabel := NewLabel;
  { The first instruction of the process will be a jump over the data we will
    insert. This data will be in the program memory six bytes following
    the start address of the process. }
Emit(JmpAlways,StartLabel);
  { Insert data into the code for use by the OS }
MemSize := DefaultStack + VarAddr[BlockLevel];
  { First, a long word indicating stack and variable size }
Emit(DecWord,(MemSize and $fff0000) div $10000) and $fff);
Emit(DecWord,MemSize and $fff);
  { Next, a word indicating the number of parameters }
Emit(DecWord,NumParm);

  { Next, for each parameter insert a word indicating the
    type of the parameter followed by a long word indicating the size of
    the parameter. }
for i := FirstParm+NumParm - 1 downto FirstParm do begin
  case ParamTable^[i].IsVar of
    ValPtype :
      case TypeTable^[ParamTable^[i].TypePointer].TypeDesc of
        RecordType,
        ArrayType : Emit(DecWord,$0001);
        ChannelType : Emit(DecWord,$0101);
        SharedType : Emit(DecWord,$0201);
        ResourceType : Emit(DecWord,$0301);
        else Emit(DecWord,$0000);
        TypSize := TypeTable^[ParamTable^[i].TypePointer].TypeSize;
      Emit(DecWord,(((TypSize and $fff0000) div $10000) and $fff));
      Emit(DecWord,(TypSize and $fff));
    end;
    VarPtype : Emit(DecWord,$0002);
    RetPtype : Emit(DecWord,$0003);
  end;
TypSize := TypeTable^[ParamTable^[i].TypePointer].TypeSize;
Emit(DecWord,(((TypSize and $fff0000) div $10000) and $fff));
Emit(DecWord,(TypSize and $fff));
224

end;
{ Now, place the start label - this is where the process code begins. }
Emit(declabel,StartLabel);
ForgetLabel(StartLabel);
{ Emit code to start the process by initializing variable and parameter
    pointers. }
Emit(StartProcess,VarAddrs[BlockLevel]);
if LexSym.Token <> begin1 then FlagError(17);
Emit(InitPF,0);
{ Parse the compound statement that comprises the code of the process. }
    ScanCmpdStat;
Emit(ClosePF,0);
Expect(semicolon1,14);
Emit(GetLabelAddress,ProcessLabel);
Emit(EndProcess,0);
{ Return to the previous block level. }
EndProcessLevel;
EndBlock;
Expect(end1,13);
end;

4.5.2.2.3 Variable Declaration Parsing

The BNF specification of a variable declaration is as follows:

<variable declaration> == var <var list> ; <type> ; [ <var list> ; <type> ; ]*

<var list> == <identifier> [ , identifier ]*

The parsing of variable declarations involves the association of a variable
name, or identifier, with a type identifier and a memory address. The type
identifier is the index of the element in the type table corresponding to the data
type. This is the number returned as a var parameter by the ScanType procedure.

Each variable declaration takes place at a given block level. Variables
declared in the main program body are given a block level of zero. Variables declared in a procedure declared in the main program body are given a block level of one. Variables declared in a nested procedure are given a block level of two, and so on. Variables declared within a given block level are stored sequentially in memory, as described in section 4.4.1 The BPL Compiler Virtual Stack. An array of variable address offsets is maintained by the compiler. Each element within the array corresponds to a block level of the program being compiled. At the beginning of each block level, the appropriate element within the array is set to zero. As each variable is declared within a given block, the variable is given an offset equal to the current value of the appropriate element within the address offset array. Then, the size of the variable is added to the element of the address offset array so that the next variable will be stored immediately following the first variable in memory.

The procedure for scanning a variable declaration is shown below:
procedure ScanVarDeclaration;
var
  IdList : array[1..64] of integer;
  i
  IdCount,
  VarTypePtr : integer;
begin
  Get; {var}
  while LexToken.Sym = identifier1 do begin
    { Initialize the identifier count variable. }
    IdCount := 1;
    { Save the name of the first identifier.}
    Get;
    { Read in the names of any other identifiers. }
    while LexToken.Sym = comma1 do begin
      Get; {comma1}
      inc(IdCount);
      IdList[IdCount] := LexToken.IntVal;
      Expect(identifier1,2);
    end;
    { Check for the : }
    Expect(colon1,5);
    { Parse the type of the variable. }
    ScanType(VarTypePtr);
    { Store the variable information for each identifier.}
    for i := 1 to IdCount do begin
      { Adjust the address offset. }
      AdjustAddr(VarOffs[BlockLevel],
          TypeTabl^[VarTypePtr].AddrMask);
      { Update the variable table. }
      StoreVarData(VarTypePtr, TypeTabl^[VarTypePtr].AddrMask);
      { Update the identifier table. }
      DeclareVar(IdList[i], VarTablPtr);
    end;
    { Check for semicolon. }
    Expect(semicolon1,14);
  end;
end;

4.5.2.2.4 Procedure and Function Declaration Parsing
BPL procedures and functions are identical except for the fact that a function returns a value, while a procedure returns nothing. Therefore, the routines that parse these programming constructs are very similar. The function parsing procedure, ScanFuncDec, will be described. Where appropriate, differences between this procedure and the procedure for parsing procedure declarations will be pointed out. The syntax for procedure and function declarations are as follows:

\[
\text{<function dec>} \equiv \text{function } \text{<func name>} [\{ \text{<parameter list>} \}] ;
\]

\[
\text{<type dec>} ; \text{<declarations> <compound statement>} ;
\]

\[
\text{<procedure dec>} \equiv \text{procedure } \text{<proc name>} [\{ \text{<parameter list>} \}] ;
\]

\[
\text{<declarations> <compound statement>} ;
\]

The first step in parsing a function is to determine the name of the function and make certain the name is not already in use. Next, a code label for the function is obtained. The function table and the identifier table are updated with the appropriate information.

A new block level is entered. All function parameters and all local declarations will be declared within this new block level. If the next token in the input stream is a left parenthesis, the ScanParmDec procedure is called to scan the parameter declarations. The function table is then modified to indicate the number of function parameters and the size of the function parameters.

For a function, the next input token must be a colon. This is followed by the type declaration of the function which is parsed by calling the ScanType procedure. This type information is then added to the type table. Procedures do
not have the colon or the type declaration, so these steps are omitted when parsing procedures.

For both functions and procedures, a semicolon must be the next input token. This is followed by any local declarations. These are parsed by calling the ScanDeclarations procedure. Note that this is a recursive call, since the ScanFuncDec procedure is called by the ScanDeclarations procedure. This allows for nested functions.

Following any local declarations is the main body of the function, indicated by the reserved word "begin." At this point, the code label is declared for the function body's executable code. Note that this label number was immediately obtained at the beginning of the function parsing. This provides nested procedures and functions with the ability to recursively call the function.

After placing the label for the beginning of the function, the intermediate operation FuncStart is performed to generate the code that produces the stack frame for the function. Next, the ScanCmpdStat procedure is called to parse the main body of the function and produce the executable code. The intermediate operation FuncEnd is then performed to generate code to restore the stack frame and return to the caller. Finally, the block level is restored by calling the procedure EndBlock.
procedure ScanFuncDec;
var
    FuncLabel,
    ParmCount,
    FType : integer;
    IdDesc : IdRec;
    Found : boolean;
    Size : longint;
    IdPtrTmp : integer;
begin
    Get; {clear out the token FUNCTION} 
    { An identifier must follow the word FUNCTION. }
    if LexSym.token <> identifier1 then FlagError(2);
    { The identifier must be unique. }
    if AlreadyDef(LexSym.i1) then FlagError(101);
    { Assign a code label to the function. }
    FuncLabel :=NewLabel;
    { Add the new function to the function table. }
    StoreFuncData(FuncLabel, ParamTablPtr+1);
    { Add the function's name to the identifier table. }
    DecFunc(LexSym.i1, FuncTablPtr);
    { Start a new block level. }
    StartNewBlock;
    { Get the next token after the identifier. }
    Get;
    { Parse the parameters, if any exist. }
    if (LexSym.token = LeftPar1) then ScanParmDec(Size)
    else Size := 0;
    { Update the function table with the parameter count.}
    ParmCount := ParamTablPtr + 1
    - FuncTabl^[FuncTablPtr].FirstParam;
    FuncTabl^[FuncTablPtr].ParameterCount := ParmCount;
    { A colon must follow the parameters. }
    Expect(colon1,5);
    { Next comes the type of the function. }
    ScanType(FType);
    { Store the function type into the type table. }
    FuncTabl^[FuncTablPtr].TypePointer := FType;
    { Store the stack offset of the function result.}
    FuncTabl^[FuncTablPtr].RsltOffset := Size;
    { Update the total size of the parameters. }
    Size := Size + TypeTabl^[Ftype].TypeSize;
    FuncTabl^[FuncTablPtr].ParamSize := Size;
    { A semicolon must follow. }
4.5.2.3 Statement Parsing

All executable instructions in BPL programs are contained within statements. The main bodies of procedures, functions, processes, and programs consist of compound statements. Compound statements in turn contain nested statements. BPL contains many different types of statements, including the statements of standard Pascal and several new statements which facilitate parallel programming.

The BNF representation of a statement is as follows:

<stat> == <if stat> | <while state> | <repeat stat> | <compound stat> | <for stat> | <case stat> | <simple stat> | <parallel stat> | <shared variable stat>

<if stat> == if <expr> then <stat> [ else <stat> ]

<while stat> == while <expr> do <stat>

<repeat stat> == repeat <stat> [ ; <stat> ]* until <expr>
<compound stat> == begin <stat> [ ; <stat> ]* end

<for stat> == for <var expr> := <expr> to <expr> do <stat>

<case stat> == case <expr> of <constant expr> ; <stat> ; [ <constant expr> ; <stat> ; ]* [ else <stat> ; end

<simple stat> == <assignment stat> | <procedure call stat>

<assignment stat> == <variable expr> := <expr>

<procedure call stat> == <identifier> [ <parm list> ]

<shared variable stat> == <read request stat> | <write request stat> | <release stat>

The syntax expression for <parallel stat> is presented in section 3.2.1 The Parallel Statement. Those for <read request stat>, <write request stat>, and <release stat> are found in section 3.1.4 Shared Variables and Templates.

As with the parsing of declarations, the parsing of statements is simplified by the fact that the first token of each statement indicates the type of the statement. The procedure ScanStat is used to parse all statements. This procedure merely inspects the current input token and calls an appropriate statement parsing procedure:

```pascal
procedure ScanStat;
begin
  case LexSym.Token of
  if1 : ScanIfStat;
  while1 : ScanWhileStat;
  repeat1 : ScanRepeatStat;
  begin1 : ScanCmpdStat;
  for1 : ScanForStat;
  case1 : ScanCaseStat;
  identifier1 : ScanSmpStat;
```
The procedures called above will now be discussed with pseudo code to describe their actions.

4.5.2.3.1 If Statement Parsing

The if statement is the simplest of all control statements. It begins with the reserved word if followed by an expression. This expression must be of type boolean. After the expression is the reserved word then followed by a statement. If the boolean expression evaluates to true, the statement is executed. Optionally, the statement can be followed by the reserved word else and another statement. This statement, if present, is executed if the boolean expression evaluates to false. The procedure for parsing if statements is of the following form:
procedure ScanIfStat;
var
  ElseLabel,
  EndIfLabel : longint;
  ExprRslt : ExprDataType;
begin
  Get; {IF}
  { Parse the expression. }
  ScanExpression(ExprRslt);
  { Make certain the expression is boolean. }
  if (ExprRslt.ExprType <> BoolType) then FlagError(135);
  { Get a label number to jump to if the boolean expression is false. This
    label number will correspond to the end of the statement or to the
    beginning of an optional else statement. }
  ElseLabel := NewLabel;
  Emit(JmpFalse,ElseLabel);
  Expect(then1,52);
  { Parse the statement. }
  ScanStat;
  { Determine if the statement has an "else." }
  if LexSym.Token = elsel then begin
    Get; {ELSE}
    { Get a label number used to skip the else statement if the expression
      was true. }
    EndifLabel := NewLabel;
    Emit(JmpAlways,EndIfLabel);
    { Place the else label here. The statement will jump to this point if the
      expression was false. }
    Emit(DecLabel,ElseLabel);
    { Parse the statement. }
    ScanStat;
    { Place the label here for skipping over the else statement. }
    Emit(DecLabel,EndifLabel);
    ForgetLabel(EndIfLabel);
  end else Emit(DecLabel,ElseLabel);
  ForgetLabel(ElseLabel);
end;

The form of the code generated by this procedure depends upon whether
or not an else clause is present. For an if statement without an else clause, the
code is of the following form:
< code to evaluate the boolean expression >
JumpFalse L1
< code corresponding to the "true" statement >

L1:

An if statement containing an else clause results in generated code of the following form:

< code to evaluate the boolean expression >
JumpFalse L1
< code corresponding to the "if" statement >
JumpAlways L2
L1:
< code corresponding to the "else" statement >
L2:

For either form of if statement, execution continues with the next instruction immediately following the if statement within the executable program memory space. This point may be reached either after the execution of a jump statement after the evaluation of the "if" statement in an if statement without an else clause, or after the execution of the "else" statement in an if statement with an else clause.

The above procedure demonstrates the use of the ForgetLabel procedure. The labels contained within an if statement are only used internally. Therefore, after the labels are no longer required for referencing, they are removed from the label table using the ForgetLabel procedure. This reduces the size requirements of the label table.
4.5.2.3.2 While Statement Parsing

The while statement is another simple control statement. It begins with the reserved word while followed by an expression. This expression must be of type boolean. After the expression is the reserved word do followed by a statement. The while statement is executed by evaluating the boolean expression. If the expression is true, the statement is executed. After the statement is executed, a jump transfers control back to the beginning of the while statement. If the expression evaluates to false, a jump transfers control to the next statement. The result is a statement that is executed repetitively so long as the boolean expression remains true. The procedure for parsing while statements is of the following form:
procedure ScanWhileStat;
var
    WhileLabel,
    FinishLabel : longint;
    ExprRslt : ExprDataType;
begin
    { Obtain two labels for internal use. }
    WhileLabel := NewLabel;
    FinishLabel := NewLabel;
    { Place the first label. }
    Emit(DecLabel,WhileLabel);
    Get; {WHILE}
    { Evaluate the expression. }
    ScanExpression(ExprRslt);
    { Make certain the expression is boolean. }
    if (ExprRslt.ExprType <> BoolType) then FlagError(135);
    { If the expression is false, jump to the finish label. }
    Emit(JmpFalse,FinishLabel);
    Expect(dol,54);
    { Parse the statement. }
    ScanStat;
    { Jump back to evaluate the expression again. }
    Emit(JmpAlways,WhileLabel);
    { Place the finish label. This is jumped to when the expression is false. }
    Emit(DecLabel,FinishLabel);
    { Discard the labels. }
    ForgetLabel(WhileLabel);
    ForgetLabel(FinishLabel);
end;

The code generated for a while statement has the following form:

L1:
    < code to evaluate the boolean expression >
    JumpFalse L2
    < code corresponding to the statement >
    JumpAlways L1

L2:

As with the if statement, the while statement continues execution with the
first machine instruction following the while statement. This occurs when the while expression evaluates to false. Also, the labels contained within a while statement are only used internally. They are therefore removed from the label table using the ForgetLabel procedure.

4.5.2.3.3 Repeat Statement Parsing

The repeat statement is similar in nature to a while statement in that the repetitive execution of a statement or group of statements is controlled by the evaluation of a boolean expression. The differences are that a repeat statement evaluates the boolean expression after execution of the statement, and that the repeat statement repeats as long as the boolean expression produces a value of false. The procedure for parsing a repeat statement has the following form:
procedure ScanRepeatStat;
var
  RepeatLabel : longint;
  ExprRslt : ExprDataType;
begin
  Get;  {REPEAT}
  { Place the label for jumping to the beginning of the repeat statement. }
  RepeatLabel := NewLabel;
  Emit(DecLabel,RepeatLabel);
  { Parse the first statement within the repeat statement. }
  ScanStat;
  { As long as semicolons are encountered in the input stream, remove }  
  them and parse a statement. }
  while LexSym.Token = semicolon1 do begin
    Get; {then semicolon}
    ScanStat;
  end;
  { Check for the reserved word "until." }
  Expect(until1,53);
  { Parse the boolean expression. }
  ScanExpression(ExprRslt);
  { Make certain the expression is boolean. }
  if (ExprRslt.ExprType <> BoolType) then FlagError(135);
  { If the expression is false, jump back to the beginning of the repeat }  
  statement. }
  Emit(JmpFalse,RepeatLabel);
  ForgetLabel(RepeatLabel);
end;

The code generated for a repeat statement has the following form:

L1:
< code correspond to the first statement >
< code correspond to the second statement >
...  
< code correspond to the last statement >
< code to evaluate the boolean expression >
JumpFalse  L1

As with the if and while statements, the internal label used in a repeat
statement it removed from the label table after the parsing of the repeat statement. The code generated for a repeat statement continues executing the next statement after the repeat statement when the boolean expression evaluates to a value of true.

4.5.2.3.4 Compound Statement Parsing

A compound statement is a group of statements that is syntactically equivalent to a single statement. As an example, the if statement executes a single statement if a boolean expression is true. If it is desired for multiple statements to be executed if the expression is true, the single statement of the if statement would be a compound statement containing the multiple statements. A compound statement begins with the reserved word begin. This is followed by a group of statements, separated by semicolons. The compound statement is terminated with the reserved word end. The procedure for parsing compound statements has the following form:
procedure ScanCmpdStat;
begin
  Get; { Clear out the BEGIN. }
  { Parse the first statement. }
  ScanStat;
  while LexSym.Token = semicolon1 do begin
    { Clear out the ";" }
    Get;
    { Scan all subsequent statements. }
    ScanStat;
  end;
  Expect(end1,13);
end;

The procedure for parsing compound statements generates no labels or branch instructions. It merely parses each successive statement, expecting a semicolon between statements.

4.5.2.3.5 For Statement Parsing

A for statement is a iterative control statement in which successive values are assigned to an integer variable for each iteration. This statement begins with the reserved for followed by a variable expression for an integer variable that serves as a loop control variable. Following this are the initial and final values to be assigned to the control variable, separated by the reserved word to and followed by the reserved word do. This is followed by the statement that is executed repetitively for each successive value of the control variable, starting with the initial value and ending with the final value. The procedure for parsing for statements has the following form:
procedure ScanForStat;
var
  ForLabel : longint;
  ExprRslt : ExprDataType;
  TypePtr : integer;
begin
  Get; {for}
  { Parse the variable expression of the control variable. }
  ScanVarAddress(TypePtr);
  { Make certain it is an integer. }
  if TypeTable^[TypePtr].TypeDesc < > IntType then FlagError(210);
  { Check for the " := " }
  Expect(assign1,51);
  { Parse the initial value expression. }
  ScanExpression(ExprRslt);
  { Make certain it is an integer. }
  if ExprRslt.ExprType < > LongintType then FlagError(211);
  Expect(to1,55);
  { Parse the final value expression. }
  ScanExpression(ExprRslt);
  { Make certain it is an integer. }
  if ExprRslt.ExprType < > LongintType then FlagError(211);
  { Store the initial value into the variable. Push the address of the
    variable and the final value of the variable onto the processor stack. }
  Emit(ForPrelim,TypePtr);
  { Get the internal label number. }
  ForLabel := NewLabel;
  Expect(do1,54);
  { Place the label where the iteration occurs. }
  Emit(DecLabel,ForLabel);
  { Parse the statement. }
  ScanStat;
  { Increment the control variable. }
  Emit(ForInc,TypePtr);
  { Repeat iteration if final value not reached. }
  Emit(JmpTrue,ForLabel);
  { Remove the internal label from the label table. }
  ForgetLabel(ForLabel);
end;

The code generated for a for statement has the following form:
< code to evaluate the address of the control variable and store it in a temporary register >
< code to evaluate the initial value for the control variable and store it in a temporary register >
< code to evaluate the final value for the control variable and store it in a temporary register >
< code to store the initial value of the control variable into the control variable and push the address and final value of the control variable onto the processor stack >

L1:
< code corresponding to the statement >
< code to increment the control variable, test to see if the final value has not been exceeded >
JumpTrue L1:
< code to remove values pushed onto the processor stack >

As with code generated for other types of statements, the code generated for a for statement continues executing the next machine instruction in program memory. The internal label used by the for statement is removed from the label table following the parsing of the statement.

4.5.2.3.6 Case Statement Parsing

A case statement chooses and executes a single statement based upon the value of an expression. This statement begins with the reserved word case followed by an expression and the reserved word of. Following this are a variable number of case selections. Each case selection consists of a constant followed by a colon (:), a statement, and a semicolon (;). Optionally, the case selections can be followed by an else clause, which consists of the reserved word else followed by a statement and a semicolon. Following either the last case selection or an else clause is the reserved word end.
A case statement is executed by evaluating the expression. The result of this expression is then compared to the constant of each case selection. If a match is found, the corresponding statement of the case selection is executed followed by a jump to the end of the case statement. If no match is found and an else clause is present, the statement of the else clause is executed followed by a jump to the end of the case statement. If no match is found and no else clause is present, the case statement executes no statement and execution continues immediately after the case statement. The procedure for parsing case statements is of the following form:
procedure ScanCaseStat;
var
   EndLabel, NextLabel : longint;
   ExprRslt, CExprRslt : ExprDataType;
begin
   { Get a label for the end of the case statement. }
   EndLabel := NewLabel;
   Get; {CASE}
   { Parse the expression. }
   ScanExpression(ExprRslt);
   { Check the type of the expression. }
   if not (ExprRslt.ExprType in [LongintType, CharType, BoolType,
   EnumType]) then FlagError(208);
   { Move the result into a temporary register. }
   Emit(EvalCase,0);
   Expect(of1,8);
   { Parse each case selection. }
   while not (LexSym.Token in [else1,end1]) do begin
      { Get a label to jump to if this case selection doesn't match. }
      NextLabel := NewLabel;
      { Parse the case selection constant. }
      ScanExpression(CExprRslt);
      if not CExprRslt.IsConst then FlagError(209);
      { Make certain the type of the constant matches the type of the
      expression. }
      if CExprRslt.ExprType <> ExprRslt.ExprType then FlagError(129);
      Expect(colon1,5);
      { Generate code to branch ahead to NextLabel if we don't match. }
      Emit(CaseCmp, NextLabel)
      { Parse the statement. }
      ScanStat;
      Expect(semicolon1,14);
      { Jump to the end of the case statement. }
      Emit(JmpAlways,EndLabel);
      { Place the label here we will jump to if we don't match. This will be
      either the next case selection, an else clause, or the end of the case
      statement if there is no else clause.
      Emit(DecLabel,NextLabel);
      ForgetLabel(NextLabel);
   end;
   { Check for an else clause. }
   if LexSym.Token=else1 then begin
      Get;
      { Parse the else clause statement. }

ScanStat;
Expect(semicolon1,14);
end;
{ Place and forget the label for the end of the case statement. }
Emit(DecLabel,EndLabel);
ForgetLabel(EndLabel);
Expect(end1,13);
end;

The code generated by a case statement is of the following form:

```plaintext
< code to evaluate the case expression, move it into register #0 >
* CASE SELECTION #1
  compare constant to register #0
  JumpNotEqual L2
  < code corresponding to statement for case selection #1 >
  JumpAlways L1
L2:
* CASE SELECTION #2
  compare constant to register #0
  JumpNotEqual L3
  < code corresponding to statement for case selection #2 >
  JumpAlways L1
L3:
* REPEAT FOR EACH CASE SELECTION
  ...
Ln:
* OPTIONAL ELSE CLAUSE
  < code corresponding to statement for else clause >
L1:
```

For each case selection, the contents of the temporary register are compared to the corresponding constant. An instruction is generated to branch ahead to the next case selection if the comparison is unequal. This is followed by the code for the corresponding case selection statement, followed by a jump to the end of the case statement. The label for the next case selection is then declared. If the
expression does not match any of the constants, execution will continue at the label following the last case selection. If there is an else clause, the code for the else clause statement will immediately follow this label. Otherwise, this label will correspond to the first instruction after the case statement.

4.5.2.3.7 Simple Statement Parsing

Simple statements are identified as beginning with a user defined identifier. This identifier can correspond to a procedure name or a variable, indicating that the statement is a procedure call or a variable assignment, respectively. The procedure for parsing a simple statement is of the following form:

```
procedure ScanSmpStat;
var
  IdDescript : IdRec;
  IdFound : boolean;
begin
  { Make certain the identifier is declared. }
  GetIdDesc(LexSym.il, IdDescript, IdFound);
  { If it is not found, flag an error. }
  if not IdFound then FlagError(104);
  case IdDescript.IdTyp of
    ProcName : ScanProcCall;
    VarName : ScanAssign;
    else FlagError(60);
  end;
end;
```

This procedure generates no executable instructions. It merely determines the type of simple statement it is parsing and calls an appropriate procedure to do the parsing. The ScanSmpStat procedure first checks to see if the identifier in the
input stream is declared. If it is not, it flags an error. Otherwise, it looks to see what type of identifier it is. If it corresponds to a procedure name, the procedure to parse procedure calls is invoked. If it corresponds to a variable name, the procedure to parse variable assignments is called. If it is neither of these, an error is flagged.

The procedure for parsing procedure calls is shown below:

```pascal
procedure ScanProcCall;
var
   IdData : IdRec;
   Found : boolean;
begin
   GetIdDesc(LexSym.il, IdData, Found);
   Get; { Skip on to the parameters. }
   { Parse the parameters, pushing their values onto the stack. }
   with ProcTabl^[IdData.ValPtr] do
      ScanParams(FirstParam, ParameterCount);
   { Generate the code to call the procedure and clean up the stack after the procedure returns. }
   Emit(ProcCall, IdData.ValPtr);
end;
```

The code produced for a procedure call is of the following form:

```plaintext
< code to evaluate and push each parameter onto the stack >
< call instruction, transferring execution to the procedure's code >
< code to remove parameters from the stack >
```

The code to evaluate and push the parameters onto the stack and the code to remove the parameters from the stack is produced only if parameters are present in the declaration of the procedure to be called.

The parsing of an assignment statement begins by evaluating the left hand
side of the assignment, which is the variable to which the assignment is to be made. The code to calculate this variable's address is generated by calling the procedure ScanVarAddress. This procedure returns, as a parameter, the index into the type table of the variable whose address is calculated. After parsing the left hand side, the parser expects to encounter the assignment operator, :=.

The next step is to parse the right hand side of the assignment. The right hand side is parsed differently depending upon the type of the variable parsed for the left hand side. For structured left hand side variables, such as records or arrays, the right hand side must be a variable of the same type as the left hand side variable. The parser then generates code to copy the structured variable on the right hand side to the variable address on the left hand side.

For nonstructured left hand side variables, the parser treats the right hand side as an expression and parses it by calling the procedure ScanExpr. Nonstructure variables include string, character, boolean, integer, and real variables. The ScanExpr procedure returns a parameter that indicates the type of expression parsed. This is tested to make certain it is compatible with the type of the variable on the left hand side.

The procedure for parsing assignment statements is of the following form:
procedure ScanAssign;
var
  LTypePointer,
  RTypePointer : integer;
  ExprRslt : ExprDataType;
begin
{ Parse the left hand side. }
ScanVarAddress(LTypePointer);
Expect(assign,51);
{ Handle different types of left hand sides differently. }
case TypeTabl^ [LTypePointer].TypeDesc of
  StringType : begin
    ScanExpression(ExprRslt);
    { Store the result as a string. }
    case ExprRslt.ExprType of
      CharType : Emit(StoreCharInStr,0);
      StringType : Emit(StoreStr,LTypePointer);
      else FlagError(129);
    end;
  end;
  BoolType : begin
    ScanExpression(ExprRslt);
    { Make certain the expression is boolean. }
    if ExprRslt.ExprType <> BoolType then FlagError(129);
    Emit(Store,LTypePointer);
  end;
  CharType : begin
    ScanExpression(ExprRslt);
    { Make certain the expression is a character. }
    if ExprRslt.ExprType <> CharType then
      FlagError(129);
    Emit(Store,LTypePointer);
  end;
  ArrayType,
  RecordType : begin
    { Structured types, RHS is a variable. }
    ScanVarAddress(RTypePointer);
    { Make certain the types match. }
    if RTypePointer <> LTypePointer then
      FlagError(129);
    { Copy the right hand variable to the left. }
    Emit(CopyStruct,
      TypeTabl^ [LTypePointer].TypeSize);
  end;
end;
The parsing of variable addresses is discussed in section 4.5.2.4 Variable Address Parsing. The parsing of expressions is discussed in section 4.5.2.5 Expression Parsing.

4.5.2.3.8 Parallel Statement Parsing

A parallel statement consists of the reserved word parallel, followed by a variable number of process invocations, and terminated with the reserved word end. Process invocations consist either of a single process variable expression with its corresponding parameter list, or a parallel form of the for statement, allowing the concurrent execution of an array of processes. The procedure for parsing parallel statements is of the following form:
procedure ScanParallelStat;
begin
Get; { PARALLEL }
{ Push a longword = -1 to mark the last process. }
Emit(IntConstant,-1);
Emit(PushVal,LongIntTypePtr);
{ Parse each process invocation. }
while LexSym.Token in [identifier,for] do begin
  case LexSym.Token of
    identifier : ScanProcessCall;
    for : ScanParallelForStat;
    else FlagError(-5);
  end;
{ Processes should be separated by semicolons. }
if LexSym.Token <> end then Expect(semicolon,14);
end;
Expect(end,13);
{ Generate the call to the OS to execute the processes. }
Emit(ParallelCall,0);
end;

The procedure ScanProcessCall parses the address of a process variable followed by the parameter list for the process. Each parameter is evaluated, with the result pushed onto the processor stack. The stack contents are then copied into the space of the corresponding process variable. The stack contents include the parameters as well as a pointer to the previous process variable. After the stack contents are copied, the parameters are removed from the stack and the pointer to the previous process variable on the stack is replaced with the address of the current process variable. Prior to the first call to ScanProcessCall, a long word value of negative one is pushed onto the stack to serve as the first previous process variable pointer.

The procedure ScanParallelForStat is identical to the procedure ScanForStat,
except that it parses a process invocation by calling the procedure ScanProcessCall rather than parsing a statement by calling the procedure ScanStat.

The last action taken by the ScanParallelStat procedure is to generate the code for a call to the operating system in order to invoke each concurrent process. This code removes from the stack the address of the last process variable and passes it to the operating system as a parameter. The operating system uses this parameter as a pointer to the first item in a forward linked list. Each item in the linked list is the space allocated in the calling process' variable memory for each process variable being invoked. Within each process variable is the following:

1. The address of the code for the process.
2. A status word indicating if the process is idle, executing, or completed.
3. Copies of parameter data pushed to the stack prior to the invocation of the process. For var and ret parameters, this is the address of the parameter variable.
4. A pointer to the next process variable in the linked list. This pointer is negative one for the last process on the list.

The operating system will step through this list and invoke each process with the appropriate parameters. The end of the list is reached when the pointer to the next process variable is negative one.

4.5.2.3.9 Shared Variable Statement Parsing

There are three different types of shared variable statements. These are the RequestRead statement, the RequestWrite statement and the Release statement. These statements are parsed by the procedure ScanSharedVarStat. This procedure
begins by inspecting the current input token to see if the current input symbol is RequestRead, RequestWrite, or Release. A case statement is used to make this determination and parse accordingly.

The RequestRead and RequestWrite statements are nearly identical in syntax and in function, allowing the same parsing routine to be used for both. The parser first makes note of whether the statement is a RequestRead and stores the result in a boolean variable. This is used at a later point to select the correct intermediate code operation. The next step is to parse the expressions for the shared variable and the template variable. These are checked to make certain that they match in type. Based upon the type of the template element, the parser can determine the number of array indices to be specified. The parser expects either a pair of integer expressions, separated by two consecutive periods, or an asterisk. If an asterisk is encountered, the entire row is selected. After all array index ranges are determined, the parser generates the intermediate code operation to complete the request operation.

The parsing of a Release statement is much simpler. It involves parsing the variable expression representing the template variable followed by executing the release intermediate code operation. The procedure for parsing shared variable statements is as follows:
procedure ScanSharedVarStat;
var ShrPtr, TempPtr, TypePtr : integer;
  DoRead : boolean;
  i,j : word;
begin
  case LexSym.Token of
    RequestReadl,
    RequestWrite1 : begin
      { Note whether this is a RequestRead. }
      DoRead := LexSym.Token = RequestRead1;
      Get;
      Expect(LeftParl,9);
      { Parse the address of the shared variable. }
      ScanVarAddress(ShrPtr);
      { Make certain its a shared variable. }
      if TypeTabl^ [ShrPtr].TypeDesc <> SharedType
        then FlagError(402);
      Expect(commal,20);
      { Parse the address of the template. }
      ScanVarAddress(TempPtr);
      { Make certain its a template. }
      if TypeTabl^ [TempPtr].TypeDesc <> TemplateType
        then FlagError(403);
      { Make sure the template and shared variables
        are of the same type. }
      if TypeTabl^ [ShrPtr].SubType <>
        TypeTabl^ [TempPtr].SubType
        then FlagError(404);
      { Generate the read or write request. }
      if DoRead then Emit(ReqRead,TempPtr)
      else Emit(ReqWrite,TempPtr);
      TypePtr := TypeTabl^ [TempPtr].SubType;
      { Get the values for each array index. }
      for i := 1 to ArrayDepth(TypePtr) do begin
        Expect(commal,20);
        { If an "*" is used, select the entire row. }
        if LexSym.Token=timesl then begin
          Get;
          Emit(IntConstant,TypeTabl^ [TypeTabl^ [TypePtr].p1].p1);
          Emit(IntConstant,TypeTabl^ [TypeTabl^ [TypePtr].p1].p2);
        end else begin
          ScanArrayIndex(TypePtr);
        end
      end;
  end;
end;
Expect(DotDot1,22);
ScanArrayIndex(TypePtr);
end;
Emit(TempRangeStore,i);
TypePtr := TypeTabl TypePtr.p2;
end;
Expect(RightPar1,4);
{ Complete the operation. }
Emit(FinishTemplateOp,TempPtr);
end;

Release1 : begin
Get;
Expect(LeftPar1,9);
{ Parse the address of the template variable. }
ScanVarAddress(TempPtr);
{ Make certain its a template variable. }
if TypeTabl TempPtr.TypeDesc <> TemplateType
  then FlagError(403);
Expect(RightPar1,4);
{ Do the operation. }
Emit(TempRelease,TempPtr);
end;
end;

4.5.2.4 Variable Address Parsing

The parsing of a variable address is performed by calling the procedure
ScanVarAddress. This procedure parses input tokens which correspond to a
variable address while invoking intermediate code operations that calculate the
actual address of the variable. Upon completion of a call to the ScanVarAddress
procedure, the address of the input variable expression becomes the top operand
on the virtual operand stack. Additionally, the ScanVarAddress procedure returns,
as a var parameter, an integer index into the type table that indicates the type of
the parsed variable expression.
BPL variable expressions can be very complex due to the variety of possible data types. As an example, a variable expression could begin with an identifier which is the name of a variable whose type is a record. If this identifier is not followed by a period, the parsing of the variable expression is complete, and the type of the variable expression is the record type. If this variable name is followed by a period, the next input token must be an identifier that is the name of one of the fields of the record type. This field component of the record type has its own type, which could be, for example, another record type or an array type. This nesting of types can continue indefinitely. The parsing of variable expressions therefore is performed using an iterative routine.

A variable expression begins with an identifier. This identifier must correspond to either a variable or a parameter. The variable or parameter is declared at a specific block level which is less than or equal to the current level. The parser begins calculating the variable address by generating code to determine the base address of variables or parameters for the desired block level. The offset address for the variable or parameter corresponding to the input identifier is added to this base address to obtain the address of the variable or parameter. The ScanVarAddress procedure stores the index into the type table of this variable or parameter in a variable named TypePtr. The parser then begins the iterative analysis of the input tokens.

The iterative routine is controlled by two values: the type pointed to by the variable TypePtr and the current input token. The possible iterative types are
record types, array types, and pointer types. If the type description pointed to by TypePtr is a record type, the current input token is inspected to see if it is a period. If the current input in a period, the next input token must be an identifier that corresponds to one of the fields of the record type. This field has a corresponding offset that is added to the top operand on the virtual operand stack in order to obtain the address of the field component. Additionally, the variable TypePtr is modified so that it points to the type description of the field component. The iteration then continues. If the variable TypePtr points to a record type and the next input token is not a period, the parsing of the variable expression is complete and the type of the variable expression is stored in the variable TypePtr.

If the variable TypePtr points to an array type description, the next input token is inspected to see if it is a left bracket. If this is the case, an expression must follow, followed by a right bracket. The expression is an integer expression that selects an element of the array variable. Code is generated to multiply this integer expression by the size of the array element type, and the result is added to the top operand on the virtual operand in order to determine the address of the array element. The variable TypePtr is updated so that it points to the type description of the array element type and the iteration continues. If the variable TypePtr points to an array type and the next input token is not a left bracket, the parsing of the variable expression is complete and the type of the variable expression is stored in the variable TypePtr.

If the variable TypePtr points to a pointer type description, the next input
token is inspected to see if it is a `^` symbol. If this is the case, a long word is fetched from the address on top of the virtual operand stack. This long word replaces the previous top operand on the operand stack. The variable TypePtr is modified so that it points to the type description of the element of the pointer type. If the variable TypePtr points to a pointer type description and the next input token is not the character `^`, the parsing of the variable expression is complete and the type of the variable is pointed to by the variable TypePtr.

The iterative parsing routine is terminated if any other data type is encountered. The form of the procedure ScanVarAddress is as follows:
procedure ScanVarAddress (var VarType : integer);
var
  TypePtr : integer;
  Found : boolean;
  IdData : IdDesc;
  Done : boolean;
  FieldIndex : integer;
begin
  if LexSym.Token <> identifier1 then FlagError(2);
  { Determine what type of identifier we have. }
  GetIdDesc(LexSym.i1, IdData, Found);
  if not Found then FlagError(104);
  case IdData.IdTyp of
    VarName : begin
      { Get the base variable address. }
      Emit(PushBaseVarAddr, VarTabl ^ [IdData.ValPtr].Level);
      { Add the offset for this variable. }
      Emit(AddConst, VarTabl ^ [IdData.ValPtr].AddrOffset);
      { Get the type of the variable. }
      TypePtr := VarTabl ^ [IdData.ValPtr].VarTypePtr;
    end;
    ParamName : begin
      { Get the base variable address. }
      Emit(PushBaseParm, VarTabl ^ [IdData.ValPtr].Level);
      { Add the offset for this parameter. }
      Emit(AddConst, ParmTabl ^ [IdData.ValPtr].AddrOffset);
      { Get the type of the parameter. }
      TypePtr := ParmTabl ^ [IdData.ValPtr].VarTypePtr;
    end;
    else FlagError(59);
  end;
  Done := false;
  while not Done do begin
    case TypeTabl ^ [TypePtr].TypeDesc of
      RecordType : if LexSym.Token = Period1 then begin
        Get; { . }
        { Find out which field we encountered. }
        if LexSym.Token <> identifier1 then
            FlagError(2);
        FieldIndex := FindField(TypePtr, LexSym.Token);
      end;
      else FlagError(59);
    end;
  end;
end;
{ Add on the field offset. }

Emit(AddConstant, FieldTabl^[FieldIndex].AddrOffs);

{ Get the type of the field. }

TypePtr := FieldTabl^[FieldIndex].FTypePtr;

end else Done := true;

ArrayType : if LexSym.Token = LeftBrktl then

begin

Get; { [ }

{ Parse the index expression. }

ScanArrayIndex;

{ Multiply by the element size. }

Emit(MultConstant, ElementSize(TypePtr));

{ Add the product to the top operand. }

Emit(Add,0);

{ Get the type of the element. }

TypePtr := TypeTabl^[TypePtr].p2;

{ Get the right bracket. }

Expect(RightBrktl,12);

end else Done := true;

PointerType : if LexSym.Token = Carrotl then

begin

Get; { ^ }

{ Fetch the long word pointer. }

Emit(Fetch, LongintTypePtr);

{ Get the type of the element. }

TypePtr := TypeTabl^[TypePtr].p2;

end else Done := true;

{ For all other types, the iteration is done. }

else Done := true;
end;
end;

{ Return the variable type as a parameter. }

VarType := TypePtr;
end;

4.5.2.5 Expression Parsing

The procedure ScanExpression is used to parse all expressions encountered in BPL programs. This procedure is capable of parsing integer, real, string, character, and boolean expressions. One var parameter of type ExprDataType is
passed to the ScanExpression procedure. The type ExprDataType is a record type containing fields that indicate the type of the expression and whether or not the expression is a constant.

As with other parsing routines, ScanExpression is a recursive descent procedure. BPL expressions are described by a hierarchy of definitions where expressions consist of signed expressions, signed expressions consist of terms, and terms consist of factors. Through the use of parenthesis, factors can consist of nested expressions. In addition to simplifying the parsing of expressions, this hierarchy of definitions is useful in that it enforces the algebraic rules for operator precedence. The BNF definitions for the expression and its components are shown below:

\[
\begin{align*}
<\text{expr}> & = <\text{sexpr}> \ [ \ <\text{cmp op}> \ <\text{sexpr}> ] \\
<\text{cmp op}> & = = | <\text{op} > | < | > | \ <= | \ >= \\
<\text{sexpr}> & = = <\text{term}> \ [ \ <\text{add op}> \ <\text{term}> ]^* \\
<\text{add op}> & = = \pm | : | \text{OR} \\
<\text{term}> & = = <\text{factor}> \ [ \ <\text{mult op}> \ <\text{factor}> ]^* \\
<\text{mult op}> & = = \ast | \div | \text{mod} | \text{and} \\
<\text{factor}> & = = <\text{constant}> | <\text{function call}> | <\text{variable reference}> | <\text{parenthetic expression}> \\
<\text{parenthetic expression}> & = = ( <\text{expr}> )
\end{align*}
\]

The ScanExpression procedure calls the procedure ScanSExpr to parse and generate code for signed expressions. The procedure ScanSExpr calls the
procedure ScanTerm to parse and generate code for terms. The procedure ScanTerm calls the procedure ScanFactor to parse and generate code for factors. Because each procedure generates code for the evaluation of each component of the expression, the algebraic rules of operator precedence are enforced. For example, the procedure ScanSExpr would call the procedure ScanTerm to evaluate a term. ScanTerm would generate the code required for any multiplicative operators before returning to ScanSExpr. ScanSExpr could then generate the code required for any additive operators, but this code would follow the code produced for the multiplicative operators. Therefore, the multiplicative operations are given precedence over the additive operators, which is correct.

The virtual operand stack maintained by the intermediate code processor requires all expressions to be translated to Reverse Polish Notation, or RPN. In this format, a binary operation is expressed by first evaluating the two factors followed by the execution of the binary operator. This translation is readily performed by the expression parsing procedures. As an example, consider the procedure ScanTerm. This procedure begins by calling the procedure ScanFactor to parse a factor. When ScanFactor returns, the parsed factor is the top operand on the virtual operand stack. If the next token in the input stream is a multiplicative operator, ScanTerm temporarily stores this token and calls ScanFactor to parse the next factor. When this second call to ScanFactor returns, the second factor becomes the top operand on the virtual operand stack and the first factor is shifted down to become the second operand on the virtual operand stack. The ScanTerm
procedure then examines the value of the stored multiplicative operator and invokes the appropriate intermediate code operation to generate the corresponding machine instructions. The ScanTerm procedure continues to call ScanFactor as long as the next input token is a multiplicative operator. Before each call to ScanFactor, ScanTerm temporarily stores the multiplicative operator; after each call to ScanFactor, ScanTerm invokes the intermediate code operation corresponding to the multiplicative operator.

The procedure ScanSExpr is functionally similar to the ScanTerm procedure. It begins by calling the ScanTerm procedure to parse a term and generate the corresponding code. It then repeatedly calls ScanTerm so long as the next input token is an additive operator. Before each call to ScanTerm, ScanSExpr temporarily stores the additive operator; after each call to ScanTerm, ScanSExpr invokes the intermediate code operation corresponding to the additive operator.

Finally, the procedure ScanExpr is functionally similar to the ScanSExpr and ScanTerm procedures. It begins by calling the ScanSExpr procedure to parse a signed expression and generate the corresponding code. It then repeatedly calls ScanSExpr so long as the next input token is a comparison operator. Before each call to ScanSExpr, ScanExpr temporarily stores the comparison operator; after each call to ScanSExpr, ScanExpr invokes the intermediate code operation corresponding to the comparison operator.

The following code sample shows the form of the procedures for parsing expressions. In order to improve the clarity of these procedures, no type checking
is performed; all factors are assumed to be either integer constants or nested parenthetic expressions:

```pascal
procedure ScanFactor;
begin
  { See what kind of factor it is. }
  case LexSym.Token of
    IntConstant1 : Emit( IntConst, LexSym.i1 );
    LeftParen1 : begin
      Get; { '(' }
      ScanExpr;
      Expect(RightParen1,4);
      end;
    else FlagError(58);
    end;
end;

procedure ScanTerm;
var
  TokenSave : LexOutput;
begin
  { Parse the leading factor. }
  ScanFactor;
  { Continue while multiplicative operators. }
  while LexSym.Token in ['*','/','div','mod','and'] do begin
    { Save the operator. }
    TokenSave := LexSym;
    Get;
    { Parse the next factor. }
    ScanFactor;
    { Generate the code for the operator. }
    case TokenSave.Token of
      '*'  : Emit(Mult,0);
      '/'  : Emit(Div,0);
      'div' : Emit(IntDiv,0);
      'mod' : Emit(Modulo,0);
      'and' : Emit(And,0);
      end;
  end;
end;
```
procedure ScanSExpr;
var
  TokenSave : LexOutput;
begin
  { Parse the leading term. }
  ScanTerm;
  { Continue while additive operators. }
  while LexSym.Token in ['+', ',', 'or'] do
    begin
      { Save the operator. }
      TokenSave := LexSym;
      Get;
      { Parse the next term. }
      ScanTerm;
      { Generate the code for the operator. }
      case TokenSave.Token of
        '+' : Emit(Add,0);
        '.' : Emit(Subtract,0);
        'or' : Emit(Or,0);
      end;
    end;
  end;
end;
procedure ScanExpr;
var
  TokenSave : LexOutput;
begin
  { Parse the leading signed expression. }
  ScanSExpr;
  { Continue while comparison operators. }
  while LexSym.Token in ['=', '<>', '>', '<', '<=', '>='] do
  begin
    { Save the operator. }
    TokenSave := LexSym;
    Get;
    { Parse the next signed expression. }
    ScanSExpr;
    { Generate the code for the operator. }
    case TokenSave.Token of
      '=' : Emit(CmpEqual,0);
      '<>' : Emit(CmpNotEqual,0);
      '<' : Emit(CmpLessThan,0);
      '>' : Emit(CmpGreaterThan,0);
      '<=' : Emit(CmpLessEq,0);
      '>=': Emit(CmpGreaterEq,0);
    end;
  end;
end;
end;

Note that the parsing of expressions is recursive. If a left parenthesis is encountered while parsing a factor, the ScanFactor procedure recursively calls the ScanExpr procedure. When parsing a nested expression, the ScanFactor inspects the token following the expression to make certain it is a right parenthesis. In this manner, the parser enforces the matching of each left parenthesis with a right parenthesis.
Chapter V

Operating System Description

This chapter presents a technical overview of the structure and functionality of the BOS operating system. Written in the BPL language, BOS is highly structured and modular, making use of data structures and associated algorithmic procedures to perform operating system functions.

The BOS operating system plays an active role in the execution of BPL programs. The function of machine allocation is performed by the operating system in response to "hints" provided by the BPL compiler. The operating system allocates system resources according to the communication relationships between processes, a novel method for load balancing.

The physical attributes of parallel program components are managed by the operating system rather than by the user program. As a result, BPL programs are high level in nature. In essence, the BOS operating system allows the programmer to create programs for a virtual parallel processing machine. Such an approach simplifies the task of parallel processing programming and facilitates the scaling of the parallel processing machine.
5.1 System Overview

The BOS operating system consists of a number of concurrent processes which cooperatively share a variety of system resources. Each process is represented within the system by a process page, a data structure containing values that describe the state of a corresponding process.

Each active process in the system is stored on a system queue. Several different system queues exist within the system. One system queue contains all processes that are ready to execute. Each other system queue contains processes that are waiting for an event to occur before they can continue executing. All processes on a given system queue are waiting for the same event to occur.

While processes execute, they request and release local and global memory pages. A set of procedures and functions are used to perform these tasks in an orderly fashion.

The BOS operating system supports the execution of BPL processes and resource procedures. A set of system queues facilitate these tasks.

Through an interface to a host computer system, the operating system supports input and output operations to files and to the operator console. A system queue is used to suspend processes waiting for host interaction.

BPL processes execute using a memory translation mechanism established by the operating system. This mechanism simplifies the allocation of process memory while protecting processes from performing erroneous memory accesses.

The operating system implements the synchronized channels of the BPL
language. A set of system queues are used to maintain processes that are suspended while waiting for communication from other processes.

The complete source code for the BOS operating system is located in Appendix B.

5.2 Memory Management

The BOS operating system allocates memory in pages. Each page is 4096 bytes in size. Throughout the execution of a program, pages are allocated and released for uses such as storing executable code, creating process local data spaces, storing operating system data structures, and implementing various high level user program data structures.

Two classes of memory pages exist: global memory pages and local memory pages. Global memory pages are used to store data structures that must be accessed by multiple processes located on different processors, while local memory pages are for use only by one processor. The function GetGlobalPage is used to allocate a global memory page. The procedure ReleaseGlobalPage is used to release a global memory page. The function GetRamPage is used to allocate a local memory page. The procedure FreeRamPage is used to release local memory page.

The global and local memory allocation routines each utilize a push-down stack for allocating memory pages. During system initialization, the two stacks are initialized such that one stack entry is created for each available page of memory.
When memory pages are requested, the top entry from the appropriate stack is popped off, removing the page address from the available memory pool. When memory pages are released, the address of the released page is pushed onto the appropriate stack, adding the page address to the available memory pool.

Each memory allocation stack is maintained using two variables: an array of long integer addresses, which stores the available memory stack, and a long integer stack pointer, which points to the top entry on the stack. The memory allocation variables for local memory allocation is stored in the local memory of each processor. The memory allocation variables for global memory allocation is stored in the global system memory where they are accessible by all processors.

When a processor allocates or releases a memory page, it must have exclusive access to the appropriate memory allocation stack variables. For local memory allocation, this is accomplished by temporarily disabling interrupts. During global memory allocation, the processor performing the memory allocation temporarily locks the global memory bus, preventing other processors from accessing the shared variables.

The memory allocation routines must deal with the possibility of no memory pages being available. The local memory allocation function GetRamPage handles this situation by calling the GetGlobalPage function and actually returning a global memory page. The global memory allocation function GetGlobalPage will generate a run time error if no additional global memory pages are available. The local memory release procedure FreeRamPage must therefore inspect the address
of the memory page being released in order to properly return global memory pages allocated using GetRamPage to the global memory available memory pool stack.

Shown below are the variables and routines used for local memory management. The global memory routines are identical except that, in addition to disabling interrupts, the global memory bus is locked prior to the execution of the critical code sections.
var
PageStack : array[0..NumRamPages] of longint;
FreePagePtr : longint;

function GetRamPage:longint;
var
MemError : boolean;
begin
MemError := false; {assume no error}
DisableInterrupts; {no interrupts allowed}
if FreePagePtr > NumRamPages then MemError := true
else begin
GetRamPage := PageStack[FreePagePtr];
inc(FreePagePtr); {pop the address}
end;
EnableInterrupts; {interrupts back on}
{ If no memory is available, request a global page. }
if MemError then GetRamPage := GetGlobalPage;
end;

procedure FreeRamPage(PageAddr : longint);
begin
{ If this is a global page, call ReleaseGlobalPage. }
if (PageAddr < RamBase) OR (PageAddr >= (RamBase + RamSize))
then ReleaseGlobalPage(PageAddr)
else begin
DisableInterrupts; {no interrupts allowed}
dec(FreePagePtr); {push the page address}
PageStack[FreePagePtr] := PageAddr;
EnableInterrupts; {interrupts back on}
end;
end;

5.3 BOS Process Pages

Each program, process, or resource in a BPL program is considered by the
BOS operating system to be a process. It should be noted that, within this
chapter, the term process will refer to such a generic process. BPL processes will
be specifically referred to as such. Each active process is represented within the BOS operating system by a "process page." A process page is merely a memory page allocated using the techniques described in section 5.2 Memory Management. Therefore, each process page is 4096 bytes in size.

A process page serves two primary functions. First of all, all operating system data for a process is stored within the process page. This data is frequently examined and modified during the life of a process. This data includes information about memory allocated to the process, values of registers when the process is suspended or blocked, data used to determine whether or not to restart a blocked process, and data used by each process to perform input and output functions with the host computer system.

This operating system data requires only a small portion of the process page. The remainder of the page is used as a supervisor stack by the process. The process page is divided as follows: the operating system data for a process resides at the end of the process page; the supervisor stack for the process is located at the beginning of the process page and extends upward to include the last byte of memory prior to the beginning of the operating system data. The supervisor stack grows in a downward direction. The initial value of the supervisor stack pointer is the address of the process' operating system data. When the first value is pushed onto the supervisor stack, the stack pointer will be decremented, resulting in this value being stored just below the operating system data in memory. The operating system data is approximately 400 bytes in size. With the BOS memory
page size of 4096 bytes, this leaves approximately 3696 bytes of memory for the process' supervisor stack. This is more than adequate for all subroutines called within the BOS operating system.

The supervisor stack is a component of the 680x0 family of microprocessors. During the normal execution of a process, the microprocessor is in "user mode," which is a protected mode of processor operation. When executing in user mode, a process cannot execute any privileged instructions. Additionally, a process executing in user mode cannot access any memory other than its own private memory. The user mode of operation therefore allows the operating system to protect critical system components and other processes when a process is executing.

The operating system, on the other hand, executes in "supervisor mode." Supervisor mode is a privileged mode of operation in which the processor can execute any instruction or access any memory. BPL programs invoke operating system functions by executing a TRAP instruction; this instruction switches the processor from user mode to supervisor mode and begins execution of a trap handling software function which, in this case, performs an operating system function. At the completion of each operation system function, an RTE instruction is executed, switching the processor back into user mode.

The 680x0 family of microprocessors provides separate stack pointers for use in user mode and in supervisor mode. Therefore, one stack is used when a BPL program executes in user mode, and a different stack is used when the BPL
program calls an operating system function and switches into supervisor mode.

The BOS operating system properly maintains the different stack pointer registers so each process is always using the appropriate stack memory area. At any time during the execution of a BOS process, the supervisor stack pointer points to a location with the process page for that process.

Below is shown the data type declaration for the operating system data stored within the process page. A description of each field will follow.

```pascal
ProcessData = record
  A5Value,
  A6Value,
  RetAddrValue,
  PrevPtr,
  NextPtr,
  QDataPtr,
  StackStore,
  PPageBusy,
  ProcessParamAddr,
  ProcessParentID,
  ProcessParentAddr,
  ResourceNext2Exec,
  ResourceNextWait,
  ResourceEndParms,
  ResourceParentA7,
  CurrentHandle,
  NextSelectNum,
  UseMMU : longint;
  IObuffer : string;
  MMUPages : LevelCDescTable;
end;

PProcessData = ^ProcessData;
```

The first three fields, A5Value, A6Value, and RetAddrValue, are used when creating a new process and preparing it for execution. All suspended BOS
processes maintain a supervisor stack containing three processor values: the value of the A5 register, the value of the A6 register, and the value of the next address for execution. Another field, StackStore, is used to store the value of the supervisor stack pointer. When a suspended process is to be started, the following steps are performed:

1. Load the supervisor stack pointer with the value contained in the field StackStore.
2. Pop the A5 and A6 registers from the stack, in that order.
3. Execute an RTS instruction, which removes an address value from the stack and continues execution at that address.

In order to create a new process, the operating system initializes the fields A5Value, A6Value, and RetAddrValue to the initial value of the A5 register, the initial value of A6 register, and the initial address for execution, respectively. The field StackStore is initialized to contain the address of the field A5Value. Because the fields A5Value, A6Value, and RetAddrValue are stored in consecutive memory locations, the values stored in these fields will be popped from the stack in the correct order.

The fields PrevPtr, NextPtr, and QDataPtr are used to store the process on a BOS system queue. Each active process is contained on one system queue. A variety of different queues exist in the BOS operating system, including a queue for processes ready and waiting to execute, a queue for processes waiting to perform I/O operations with the host, and a queue for processes waiting to execute
within resources. BOS system queues are discussed in section 5.4 BOS System Queues.

The fields ProcessParamAddr, ProcessParentID, and ProcessParentAddr are valid only if the process page represents a BPL Process. Prior to the execution of a process, the operating system creates a stack frame for the process containing the process parameters. The operating system stores the address of the first parameter in the field ProcessParamAddr. When the process terminates, the operating system uses this value to copy back any var or ret parameters to the parent process. The ProcessParentID field contains a value that is used to identify the parent process. The low-order eight bits of this value are the physical processor number. The high-order twenty-four bits of this value are the address of the process page for the parent process. Finally, the field ProcessParentAddr is the address within the user address space of the parent process of the process variable corresponding to this child process. This value is used by the operating system to copy parameters back to the parent process. These fields are discussed further in section 5.5 Process Execution.

The fields ResourceNext2Exec, ResourceNextWait, ResourceEndParms, ResourceParentA7 are valid only if the process page represents a BPL resource. The ResourceNext2Exec and ResourceNextWait fields are used to implement a fair arbitration protocol when multiple concurrent processes attempt to execute within a common resource. The ResourceEndParms field is used by the operating system when copying parameters to and from the calling process. The ResourceParentA7
field is used by the operating system to temporarily store the stack pointer of the calling process during execution of a resource entry procedure. These fields are discussed further in section 5.6 Resource Execution.

The fields CurrentHandle and IOBuffer are used to improve the efficiency of host output operations. The CurrentHandle field contains the file handle of the last output operation. The IOBuffer field is a string variable in which multiple successive output strings are concatenated. By using these fields, the operating system is able to group several output operations into a single host output request. These fields are discussed further in section 5.7 Host I/O.

Finally, the fields UseMMU and MMUPages are used by the operating system to program the 68030 Memory Management Unit. This mechanism is used to isolate the address spaces of all system processes from one another. The field UseMMU contains a non-zero value if the associated process uses the memory management unit. All BPL processes and resources use the memory management unit. BOS System processes, on the other hand, do not use the memory management unit, and therefore have a zero value stored in the UseMMU field. The MMUPages field is structured variable used to contain data describing the memory translation mapping for the process. These fields are discussed further in section 5.8 System Memory Translation Management.
5.4 BOS System Queues

The BOS operating system uses system queues to keep track of all active processes. An active process is here defined to be any process currently existing within the system. An active process can be executing, ready to execute, or blocked. When a process terminates, it is removed from the system and ceases to be an active process.

BOS processes can exist in a variety of different states. The state of a process is determined by whether the process is executing, ready to execute, or blocked. If a process is blocked waiting for an event, the state of that process is further determined by the event for which the process is waiting. In the BOS system, a different system queue is maintained for each possible process state.

One "rule" strictly adhered to within the BOS operating system is that each process must always be present on one system queue. Immediately after a process is created, it is placed on a system queue. During the life of a process, the process will change its state, requiring it to be removed from one queue and placed onto another. All operating system code is carefully written so that each operation that removes a process from a queue is immediately followed by a corresponding operation to place that process onto another queue.

A system queue is represented by a variable of type QueueType. The type declaration for QueueType is as follows:
As can be seen, this is a very simple type, consisting of only three fields. The first two fields, Head and Tail, are used to maintain a doubly linked list of all processes on the queue. The Head field contains the address of the first process on the queue, while the Tail field contains the address of the last process on the queue. The other field, PCount, is simply a count indicating the number of processes on the queue.

A variety of procedures and functions are used to manipulate the BOS system queues. The BOS operating system uses these routines exclusively when dealing with system queues. An important feature of each of these routines is that all code is written such that multiple concurrent processes can safely access system queue. For example, a system interrupt handler can call any BOS system queue procedure or function at any time without interfering with other processes working with the same system queue. Each of these procedures and functions will now be discussed.

The procedure InitQ is declared as follows:

```pascal
procedure InitQ(var Q : QueueType);
```

This procedure is used to initialize a system queue. This procedure is called once
for each system queue during the initialization of the operating system. InitQ sets the PCount field of the queue to zero, indicating that no processes are on the queue. Initially, the fields Head and Tail are initialized so that they point to each other.

The procedure PutOnQ is declared as follows:

```
procedure PutOnQ(var Q : QueueType; ProcessPage : longint);
```

This procedure is used to place a process onto a specific system queue represented by the parameter Q. The second parameter, ProcessPage, is the address of the process page for the process to be placed onto the queue. The queue fields Head and Tail are modified in order to place the process at the end of the queue. Additionally, the field NextPtr and PrevPtr within the process page of the process being added are updated in order to maintain a doubly linked list. Finally, the queue field PCount is incremented to reflect the presence of the new process on the queue.

The doubly linked list is maintained in the following manner. The queue field Head contains the address of the field NextPtr for the first process on the queue, while the queue field Tail contains the address of the field PrevPtr for the last process on the queue. The process field NextPtr for the first process on the queue contains the address of the queue field Head, while the process field PrevPtr for the last process on the queue contains the address of the queue field Tail. The following illustrates the values of the queue and process fields for a system queue.
containing a single process:

\[
\begin{align*}
\text{Queue.Head} &= \text{ADDR Process.NextPtr} \\
\text{Process.NextPtr} &= \text{ADDR Queue.Head} \\
\text{Process.PrevPtr} &= \text{ADDR Queue.Tail} \\
\text{Queue.Tail} &= \text{ADDR Process.PrevPtr}
\end{align*}
\]

When more than one process is present on a system queue, pointers are maintained between consecutive processes. Assume that Process B immediately follows Process A on a system queue. The following illustrates the pointer values linking these two processes:

\[
\begin{align*}
\text{Process A.PrevPtr} &= \text{ADDR Process B.NextPtr} \\
\text{Process B.NextPtr} &= \text{ADDR Process A.PrevPtr}
\end{align*}
\]

Therefore, a system queue with, for example, three processes present would have the following combination of field pointer values:

\[
\begin{align*}
\text{Queue.Head} &= \text{ADDR Process #1.NextPtr} \\
\text{Process #1.NextPtr} &= \text{ADDR Queue.Head} \\
\text{Process #1.PrevPtr} &= \text{ADDR Process #2.NextPtr} \\
\text{Process #2.NextPtr} &= \text{ADDR Process #1.PrevPtr} \\
\text{Process #2.PrevPtr} &= \text{ADDR Process #3.NextPtr} \\
\text{Process #3.NextPtr} &= \text{ADDR Process #2.PrevPtr} \\
\text{Process #3.PrevPtr} &= \text{ADDR Queue.Tail} \\
\text{Queue.Tail} &= \text{ADDR Process #3.PrevPtr}
\end{align*}
\]

The procedure TakeOffQ performs the task of removing a process from a specified system queue. This procedure is declared as follows:
procedure TakeOffQ(var Q : QueueType; PPage : longint);

This procedure takes care of all pointer manipulations required to remove a process from a queue. The process to be removed from the queue is designated by the parameter PPage, which is the address of the appropriate process page. The pointers within this process page are updated so as to remove the hole in the doubly linked list caused by the process removal. Additionally, the queue field PCount is decremented to reflect the removal of the process.

The procedure JoinQFlick performs two related tasks: joining a system queue and relinquishing the processor. This procedure is declared as follows:

procedure JoinQFlick(var Q : QueueType; ReleaseCondValPtr : longint);

The first parameter, Q, represents the system queue the process will join. The second parameter, ReleaseCondValPtr, is a long integer value that can be used to provide additional information indicating why a process is stored on a queue. This value is stored within the operating system process data in the field QDataPtr. This value takes on different meanings for the various system queues. For example, processes waiting on the system queue ResourceWaitQ are waiting to execute within a busy BPL resource. For these processes, the value of ReleaseCondValPtr is the global address of the resource for which each process is waiting. The system queue ReadyQ, on the other hand, contains all processes which are ready to execute. These processes have no condition or event that they wait for. Therefore, the value of ReleaseCondValPtr for these processes is ignored.
For other queues, the value of ReleaseCondValPtr might be a pointer to a data structure describing a more complicated event’s data.

Unlike the procedure PutOnQ, the procedure JoinQFlick does not specify a process page. Instead, the process to join the system queue is the currently executing process. Therefore, the process page can be determined by inspecting the value of the stack pointer.

The procedure Flick is similar to the procedure JoinQFlick. The Flick procedure is declared as follows:

```
procedure Flick(ReleaseCondValPtr : longint);
```

The difference between Flick and JoinQFlick is that no system queue is specified for the current process to join. This procedure causes the process to relinquish control of the processor without joining a system queue. The process must already be on a queue for this procedure to function properly.

The procedure LeaveQ is similar to the procedure TakeOffQ. The LeaveQ procedure is declared as follows:

```
procedure LeaveQ(var Q : QueueType);
```

Unlike the procedure TakeOffQ, no process is specified for the procedure LeaveQ. The LeaveQ procedure is used for removing the current process from the specified process. The process page address is determined by inspecting the value of the
stack pointer.

The function QReady is a boolean function with the following declaration:

```pascal
function QReady(var Q : QueueType) : boolean;
```

This function inspects the specified system queue and returns a value of true if a process is present on the queue or a value of false if the queue is empty.

The functions GetFirstProcess and GetNextProcess are used to examine each process on a queue. These functions are declared as follows:

```pascal
function GetFirstProcess(var Q : QueueType) : longint;
function GetNextProcess(var Q : QueueType; StartPoint : longint) : longint;
```

The GetFirstProcess function returns the address of the process page for the first process page on the specified system queue. The GetNextProcess function returns the address of the process page of the next process on the specified queue following the process whose process page address is specified by the parameter StartPoint. Either of these functions returns a value of negative one if no more processes are present on the system queue. GetFirstProcess would therefore return a value of negative one if the specified queue is empty, while GetNextProcess would return a value of negative one if the process specified by the parameter StartPoint is the last process on the specified queue.

These functions are used examine the processes on a specific system queue in the proper order. Consider, for example, a routine to search for a process
waiting to execute within a specified resource. All processes waiting to execute within resources are located on the system queue ResourceWaitQueue. The routine to search for a process waiting for access to a specified resource would begin by calling the function GetFirstProcess using ResourceWaitQueue as a parameter. If this function returns a value of negative one, the search is aborted. Otherwise, the value returned is the address of the process page for a process waiting for access to a resource. The field QDataPtr of this process page would next be inspected to see if it contains a value corresponding to the address of the specified resource. If a match is found, the search is complete. Otherwise, GetNextProcess is called in order to obtain the address of the process page for the next process on the queue. This search is repeated until either a matching process is found or the GetNextProcess function returns a value of negative one.

The queue manipulations used to control resource execution are described in more detail in section 5.6 Resource Execution.

The procedure ExecQ is used to execute the first process on a queue. The declaration for ExecQ is as follows:

```pascal
procedure ExecQ(var Q : QueueType);
```

The ExecQ procedure obtains the address of the first process on the process queue. Within the process page for that process, the field StackStore can be read. As described previously, when a suspended process is to be started, the following steps are performed:
1. Load the supervisor stack pointer with the value contained in the field StackStore.

2. Pop the A5 and A6 registers from the stack, in that order.

3. Execute an RTS instruction, which removes an address value from the stack and continues execution at that address.

The procedure Relinquish is used by an active unblocked process to yield the processor to another ready process. This procedure is called after non-blocking I/O operations in order to fairly distribute processor usage. Additionally, the Relinquish procedure is called from within the periodic timer interrupt service routine. In this simple manner, time slicing is achieved. The Relinquish procedure performs two procedure calls. First, the LeaveQ procedure is called to remove the active process from the ReadyQ, the system queue containing unblocked processes. Next, the JoinQFlick procedure is called to place the process back onto the ReadyQ and release the processor. By immediately following LeaveQ with JoinQFlick, the process is removed from the beginning of the queue and returned to the end of the queue, moving it to the "end of the line" for ready processes.

The following is a list of all BOS system queues, along with a brief description of each:

**ReadyQ**: All processes contained on the ready queue are ready to execute. There are a variety of different classes of processes which can be placed on this queue, including newly loaded programs, child processes created during the execution of a program, previously executing processes that were interrupt by the periodic timer, and previously blocked processes that have been removed from
other queues.

**ParentProcQ**: All processes on this queue are parent processes which have created one or more child processes using a BPL parallel statement. When a parent spawns children, it allocates a global memory page in which children indicate when they have terminated. When a parent process joins the queue ParentProcQ, the address of this global page is passed as the ReleaseCondValPtr parameter. This enables the operating system to easily check all suspended parent processes for terminated child processes. After all child processes have terminated, the parent process is removed from the queue ParentProcQ and placed on the queue ReadyQ.

**ResourceWaitQ**: Processes waiting to execute within a busy resource are stored on the system queue ResourceWaitQ. When a process places itself on the queue ResourceWaitQ, the address of the resource process page is passed as the ReleaseCondValPtr parameter. This procedure is described in more detail in section 5.6 Resource Execution.

**OpenChanQ**: All BOS channel operations are synchronous. When a process attempts to open a channel for reading, the process is blocked until another process opens the same channel for writing. Similarly, when a process attempts to open a channel for writing, the process is blocked until another process opens the same channel for reading. All such blocked processes are stored on the system queue OpenChanQ. This procedure is described more fully in section 5.9 BPL Channel Operations.
SendChanQ and RcvChanQ: These queues are used to store processes which are blocked by a channel operation. When a sending process attempts to send a message, if a corresponding process is not ready to receive, the sending process will be suspended and stored on the system queue SendChanQ. Conversely, a receiving process will be suspended on the system queue RcvChanQ until a corresponding process is ready to send. Channel sending operations are described in section 5.9 BPL Channel Operations.

EtherQ: This system queue is for suspending processes waiting for messages from the host PC via the Ethernet interface. All file and console I/O functions are performed using a sequence of Ethernet messages. When a process completes an I/O operation, it removes itself from the queue EtherQ and joins the queue ReadyQ. Host I/O procedures are discussed in section 5.7 Host I/O.

5.5 Process Execution

The execution of processes is heavily dependant upon the system queue mechanism described in section 5.4 BOS System Queues. When the system is powered on, the operating system initializes all system queues on each processor to be empty. Following this initialization, the operating system on each processor begins executing the following instructions:

```
ProgramDone := false;
repeat
    CheckForNewProcesses;
    CheckForFinishedProcesses;
    CheckForReadyResources;
```
CheckForReadyChannelOperations;
if QueueReady(ReadyQ) then
    ExecQ(ReadyQ);
until ProgramDone;

The first statement initializes the boolean variable ProgramDone to false. The operating system, while executing a program, can terminate the program by assigning a value of true to this variable. This can be done in response to an error or in order to normally terminate a completed program.

The operating system then enters a repeat loop which controls the execution of BPL programs. The first procedure in the loop, CheckForNewProcesses, looks to see if any new child processes are created for this processor; if so, these processes are moved to the ReadyQ. The next procedure call in the loop, CheckForFinishedProcesses, inspects the system queue ParentProcQ, which contains all parent processes waiting for child processes to terminate. For each suspended parent process on the system queue ParentProcQ, the CheckForFinishedProcesses procedure checks the system looking for children of that parent which have terminated. If any parent processes are encountered for which all child processes are terminated, the parent is moved from the system queue ParentProcQ and placed on the system queue ReadyQ. Child and parent process procedures will be discussed further later in this section.

The next procedure called in the loop, CheckForReadyResources, inspects the system queue ResourceWaitQ and attempts to locate any processes which are waiting to execute within available resources. If any are encountered, these
processes are removed from the system queue ResourceWaitQ and added to the system queue ReadyQ. This procedure is discussed in section 5.6 Resource Execution.

Next, the CheckForReadyChannelOperations procedure is called. This procedure inspects the system queues OpenChanQ, SendChanQ, and RcvChanQ to look for suspended processes which are ready to be resumed. If any such processes are located, any required channel manipulations are performed and the processes are moved onto the system queue ReadyQ. Channel management is discussed in section 5.9 BPL Channel Operations.

The final action in the loop executed by the operating system is to inspect the system queue ReadyQ to see if any processes are present. If so, the first process on the queue is executed using the procedure ExecQ. When the ExecQ procedure returns, the process that executed has relinquished the processor, either due to the time slicing timer interrupt or due to a voluntary release of the processor. Processes which are interrupted for time slicing are moved back onto the system queue ReadyQ. Processes which voluntarily release the processor include those which are blocked awaiting an event, and therefore have moved off the system queue ReadyQ and onto another system queue, and those which are terminated, and have removed themselves from all system queues and released any allocated system memory.

When the operating system initializes itself on each processor, all system queues are empty. The procedures contained with the repeat loop, which all
perform functions only if processes are located on specific system queues, will therefore have nothing to do. Once a single process is introduced into the system, however, all the various system queues can become active. For example, a single process can create multiple child processes, some of which might use channels to exchange messages, others that will attempt to execute within shared resources, still others that will create other child processes, and so on. The operating system on each processor node, however, will remain idle until a single parent process is created.

This single parent process is created in response to messages received from the host computer system. The host begins the execution of a program by sending, to each processor in the system, a sequence of messages containing the executable code. Each processor responds to each message by allocating any needed memory for the executable code, storing the code in the memory, and sending an acknowledge message back to the host.

After the entire program is loaded onto each processor, the host sends a message to the first processor in the system, instructing it to create a new process. The resulting process corresponds to the main body of the program and therefore will be the ancestor of all child processes. The message sent to the first processor contains the address of the main body of the program. The operating system responds to the message by calling the function CreateProcessSpace, which has the following declaration:

```pascal
function CreateProcessSpace( ProgAddr, PType : longint) : longint;
```
The function CreateProcessSpace is used by the operating system to create all new processes, including the main program process and any child processes created by the main program process or by descendants of the main program process. The two parameters, ProgAddr and PType, are used to indicate the execution address of the process and the process type, respectively. A value of zero (0) for PType indicates a main program process, while a value of one (1) indicates a child process. The longint result returned by CreateProcessSpace is the address of the process page for the newly created process.

The CreateProcessSpace function begins by allocating a page of memory to be used as a process page for the new process. It then allocates another page of memory used for performing virtual memory address translation by the 680x0 memory management unit. The BPL compiler stores information in the executable code indicating the size of local variables and stack space required by the process. The required amount of memory is allocated, with the physical address of each allocated page stored in the page used for memory address translation. The issues relating to memory address translation are discussed in section 5.8 System Memory Translation Management.

As described in section 5.3 BOS Process Pages, several values in the process page data structure must be initialized so that the process can be started. When a process is executed from a system queue, the process will be started in supervisor mode. This is appropriate, since processes join the system queue either while executing an interrupt handler or while executing an operating system function.
request, both of which execute in supervisor mode. Processes, however, execute in user mode, and the process address passed to the CreateProcessSpace function represents a user mode address. Therefore, the CreateProcessSpace function creates supervisor mode code for each process to set up the processor registers for user mode execution and perform the transition from supervisor mode to user mode.

Each process created by the CreateProcessSpace function will have its own unique code created for this transition. Because this code is less than thirty-two bytes in length, the beginning portion of the process page is a suitable and safe location for the code. Note that this area is where the stack can expand to in a downward direction. Initially, however, only a few bytes of stack space, with the result that several thousand bytes of memory at the beginning of the process page are safe to use.

The CreateProcessSpace function moves machine instruction values into the process page starting at the beginning of the page. The machine instructions first initialize the user stack pointer with the address of the user mode stack. The next instructions build up an exception stack frame consisting of the user mode process address and a system status register that will place the process in user mode. The final instruction is an RTE instruction, which reads the exception stack from and places the processor into user mode executing at the beginning of the process.

In addition to storing the initialization code into the low bytes of the process page, the process page data fields A5Value, A6Value, RetAddrValue, and
StackStore must be initialized. The initial values of address registers A5 and A6 are not important, so the fields A5Value and A6Value are initialized to zero. The reason for this is that the first task a new process performs is to initialize these registers, so that any value stored by the operating system is immediately overwritten. The field RetAddrValue is initialized to contain the address of the first instruction to be executed which, in this case, is the base address of the process page. Finally, the field StackStore must be initialized to contain the appropriate value for the stack pointer for the newly created process. The value is the address of the A5Value field of the process page.

With these values stored, the process will begin as follows: the stack pointer is loaded with the value stored in the field StackStore. Next, the address registers A5 and A6 are initialized to zero by removing the values from the stack corresponding to the data stored in the fields A5Value and A6Value, respectively. Finally, a RTS instruction removes from the stack the value stored in the field RetAddrValue, stores this value in the program counter, and forces execution to begin at this address. For newly created processes, this address is the base of the process page where the initialization instructions are stored.

The first call to CreateProcessSpace is performed in response to a message received from the host computer instructing the first processor to begin execution of the program. The value returned by CreateProcessSpace is the address of the process page corresponding to the main process of the program. This value is used to place the main process onto the system queue ReadyQ. From this process, all
other processes are spawned, providing the operating system with activity on the various system queues.

Additional processes are introduced into the system when the main process executes a parallel statement. As described in section 3.2.1 The Parallel Statement, a parallel statement is used to execute multiple concurrent child processes, suspending the parent process until all children terminate. Section 4.4.3.6 Process and Resource Calling Operations describes the code generated by the BPL compiler for the execution of a parallel statement. In summary, the compiler generates a call to the operating system requesting the execution of a group of processes. The single parameter passed for this call is the address of the first child process variable. One field of this process variable is a link pointing to the next process variable. The last child process on the linked list has a link value of negative one, indicating the end of the list.

The operating system begins the execution of the parallel statement by allocating a global memory page. This memory page will be used by the child processes to notify the parent when they are terminated. Because a global page is used, processes on any processor can notify the parent.

The operating system next steps through the linked list of child processes and creates a process page for each in the global memory. These process pages are created by calling the function CreateProcessSpace, which is described earlier in this section. The process code address passed to the CreateProcessSpace function is obtained from a field within the process variable. A value of one (1)
is passes for the PType parameter to indicate that a child process is to be created.

The call to CreateProcessSpace returns the address of the global memory page that will serve as a process page for the child process. Within this process page, the data field ProcessParentID is assigned the address of the global memory page the child process will use to notify the parent upon termination. The data field ProcessParentAddr is assigned the address of the process variable maintained by the parent for the child process. The child will pass this value back to the parent upon termination so that process parameters can be properly returned to the parent.

After these fields are initialized, the operating system calls the procedure CopyPParms in order to copy any parameters from the parent to the child. The CopyPParms procedure is used to send parameters to a child process before process execution, and to return parameters from the child process following process execution. This procedure has the following declaration:

```plaintext
procedure CopyPParms(PIAddr, PPage, ParmAddr, Direction : longint);
```

This procedure receives four longint parameters. The first parameter, PIAddr, is the address of the first executable instruction for the child process. The first instruction in a BPL process is a long jump instruction to the actual beginning of the process code. The BPL compiler stores information regarding the parameters for the child process immediately following this jump instruction. Because the long jump instruction requires six bytes, the child process parameter
information begins six bytes after the address of the process code. A two byte integer stored at this address contains the number of parameters passed to the child process. Following this two byte value are six bytes for each process parameter. The first two bytes of the six are a two byte integer representing the class of the parameter. The following classes are defined:
Class = 0 : The parameter is a value parameter with the actual value stored within the process variable.

Class = 1 : The parameter is a value parameter with a pointer to the value stored within the process variable.

Class = 2 : The parameter is a var parameter with a pointer to the referenced variable stored within the process variable.

Class = 3 : The parameter is a ret parameter with a pointer to the referenced variable stored within the process variable.

The remaining four bytes following the class identifier are a long integer representing the size of the parameter data type. Immediately following the data for the last parameter is the beginning of the executable code for the process. This executable code is the target of the long jump located immediately before the process parameter information.

The next two parameters to the CopyPParms procedure are the address of the child process page and the address of the evaluated process parameters stored in the process variable. These two addresses are used to determine the source and destination addresses when the process parameters are copied. The final parameter, Direction, is used to specify whether the parameters are to be copied to the child or back to the parent. A value of zero (0) is used to indicate that the parameters are to be copied back to the parent, while a value of one (1) indicates that the parameters are to be copied to a newly created child process.

After the parameters are copied to the data space of the child process, the procedure PlaceOnProcessor is called in order to notify a processor in the system that a new child exists for it to execute. This notification is performed using a
global memory page.

During the initialization of the operating system, a global memory page is reserved for each processor. This page is used to store a list of new processes. Initially, this page is empty. The PlaceOnProcessor procedure adds a new process to the page. Earlier in this section, the procedure CheckForNewProcesses is shown as one of the procedures called repeatedly by the operating system during the execution of a BPL program. This procedure simply checks for the existence of new processes on the global memory page allocated for that processor. When new processes are located, they are removed from the global memory page and moved onto the system queue ReadyQ. In this manner, the process begins execution on another processor.

After all child processes are initiated and placed onto processors, the parent process must wait for all children to terminate. The parent process removes itself from the system queue ReadyQ and joins the system queue ParentProcQ. While the operating system executes a program, it repeatedly calls the procedure CheckForFinishedProcesses. This procedure simply executes each process on the system queue ParentProcQ. These processes, when invoked, check for finished child processes and, if any are found, copy back any process parameters. When the last child is terminated, the parent process removes itself from the system queue ParentProcQ and joins the system queue ReadyQ. Below is shown the pseudo code executed by the parent process after the child processes are started:
LeaveQ(ReadyQ); { Leave the ready queue }
while ActiveChildren > 0 do begin
  JoinQFlick(ParentProcQ); { Relinquish the CPU }
  if "A child is done" then begin
    "Copy back the parameters from the child."
    dec(ActiveChildren); { Keep count }
  end;
  LeaveQ(ParentProcQ); { Get off queue briefly }
end;
JoinQFlick(ReadyQ); { All children are done }

This code example illustrates the system queue facilities. Initially, the process is active on the system queue ReadyQ. The process removes itself from this queue and enters the while loop. The first action taken inside the loop is to join the system queue ParentProcQ. This relinquishes the processor to execute active processes. Meanwhile, the operating system periodically executes the procedure CheckForFinishedProcesses. This procedure executes each process on the system queue ParentProcQ.

The parent process is therefore restarted, and continues executing immediately following the JoinQFlick statement. The parent checks to see if a child process is terminated. If so, the parent copies back the process parameters. Additionally, the variable ActiveChildren is decremented in order to keep track of the number of active children. Next, the parent process removes itself from the system queue ParentProcQ and returns to the beginning of the while loop. If more active children exist, the parent process immediately joins the system queue ParentProcQ and relinquishes the processor.

The parent process is then restarted the next time the operating system
executes the CheckForFinishedProcesses procedure. If additional child processes exist, the parent repeatedly removes itself from the system queue ParentProcQ and immediately joins the queue again. When the last child process is terminated, the parent process joins the system queue ReadyQ, making it ready for normal execution.

The child process performs two simple tasks when it terminates. First, it notifies the parent process that it is terminated. This is done by obtaining, from the process page of the child process, the address of the global memory page used to signal the parent upon completion. The child then stores its own address in this page for the parent to find. Next, the child process removes itself from the system queue ReadyQ. The child process then executes the Flick procedure, relinquishing the processor. Because the child does not join a system queue prior to executing the Flick procedure, the process is no longer a part of the operating system. The parent process takes care of releasing all memory allocated to the child.

The BPL language is implemented such that when a procedure or function is called, the only processor registers that must be returned unmodified are address registers A5, A6, and A7. For this reason, when a process voluntarily relinquishes the processor by calling the procedure Flick, the operating system only needs to save these registers; when Flick returns to the caller, the suspended code assumes all other register values are invalid. In order to satisfy the restarting of processes as described in section 5.3 BOS Process Pages, the operating system performs the following tasks when relinquishing the processor:
1. Push address register A6 onto the stack.
2. Push address register A5 onto the stack.
3. Move the current value of address register A7 into the process page data field StackStore.

When this sequence is performed, the operating system can restart a suspended process. The process can be suspended while executing inside several nested procedures. When restarted, all local procedural variables and parameters will maintain the same values as they did prior to suspending the process. This property is useful in that it allows suspended processes to store process state variables as local variables with local scope, rather than requiring the operating system to provide distinct storage space for such state variables.

Processes suspended due to a timer interrupt are different from those suspended voluntarily in that the interrupt can occur at any time. Whenever a process is started in user mode, the procedure TimerStart is executed just prior to restarting the process. This procedure initializes a timer which generates an interrupt in fifty milliseconds. When a process in user mode enters supervisor mode in order to request an operating system function, the procedure TimerStop is executed. This procedure stops the timer and disables the timer interrupt.

If a process executes in user mode for more than fifty milliseconds, a timer interrupt is generated. Because of the conditions under which the timer is started and stopped, the timer interrupt can assume that if the interrupted program is a user mode process, it is executing on the system queue ReadyQ. The procedure
that responds to the timer interrupt is a BPL exception procedure, as described in section 3.3.1 Exception Procedures. The code generated for an exception procedure saves all system registers upon entering the procedure and restores all system registers upon exiting.

The timer interrupt handler merely removes the current process from the system queue ReadyQ, joins the system queue ReadyQ at the end of the list, and relinquishes the processor to the operating system, allowing the operating system to restart any unblocked processes. When the interrupted process is restarted, execution resumes within the interrupt handler. The exit code for this procedure restores all system registers and returns to the interrupted code.

5.6 Resource Execution

Two issues must be considered for the execution of resource procedures. The first issue is controlling when processes are permitted to execute within the resource. The second is the transformation of the process state so that it executes within the process environment of the resource.

Each resource is created within the system as a process, complete with a process page, a memory translation table, and local ram for static variables and stack. All memory allocated to the resource, including the process page, is allocated from the global memory space. This enables all processors to access the resource. A resource variable is a single long integer value. This value is the address of the process page for the resource. When a resource is passed as a
parameter to a child process, this address is the value sent. Unlike normal processes, idle resources are not stored on system queues. They are instead maintained in the variable space of each process that declares a resource variable. When such a process terminates, the resource is removed from the system.

The process page of a resource contains two fields used to control multiple concurrent requests for execution within the resource. These fields are named ResourceNextWait and ResourceNext2Exec. When the resource is created, these fields are initialized to zero (0).

When a process wishes to execute within a resource, it calls the function TakeANumber to request an access number. The declaration of this function is as follows:

    function TakeANumber(RPage : longint) : longint;

The single parameter RPage is the address of the resource process page. The TakeANumber function returns the value of the resource process page data field ResourceNextWait. It then increments this number so that the next process to call TakeANumber for a resource will receive the next higher integer result. Concurrent accesses to the resource process page are prevented in the TakeANumber function so that each call is guaranteed to return a unique value.

The requesting process, after calling TakeANumber, is permitted to execute within the resource when the process page data field ResourceNext2Exec is equal to the value returned by TakeANumber. When a process finishes executing within
a resource, it increments the process page data field ResourceNext2Exec, granting permission to the next process waiting to execute within the resource.

Consider the example of a process requesting access to a newly created resource. The resource process page data fields ResourceNextWait and ResourceNext2Exec are both initially zero (0). Therefore, the first call to TakeANumber will return a value of zero (0), incrementing the data field ResourceNextWait to one (1). After calling TakeANumber, the process attempting to execute within the resource compares the returned value with the value of the data field ResourceNext2Exec. Because this field is initially zero (0), the process is free to execute within the resource.

If another process attempts to execute within the same resource while the first process is still executing, calling TakeANumber will return a value of one (1), while the data field ResourceNext2Exec is still zero (0). Therefore, this second process is not permitted to execute within the resource. When the first process exits the resource, it increments the data field ResourceNext2Exec, therefore granting permission to the second process to execute within the resource.

The system queue ResourceWaitQ is used to store processes which are waiting to execute within a resource. As described in the previous section, the operating system periodically executes the procedure CheckForReadyResources. This procedure inspects each process on the system queue ResourceWaitQ to see if the value of TakeANumber is the current value of the data field ResourceNext2Exec for the specified resource. All processes which satisfy this
criteria are removed from the system queue ResourceWaitQ and placed on the system queue ReadyQ. The operating system will then restart the suspended process, at which time it executes within the resource. When the process exits the resource, it increments the data field ResourceNext2Exec, allowing other processes access to the resource.

Once a process is granted permission to execute within a resource, several steps are required to properly transfer the process to the environment of the resource and, upon exiting the resource, back to the environment of the process.

The first step to be performed is to copy any value or var resource parameters from the calling process to the address space of the resource. While doing this, the starting address of the parameters is calculated to be the address of the permanent resource variables minus the total size of the parameters.

Next, code must be created and stored in the resource process page, similar to that created for new processes. This code must perform several tasks. First, it must store the stack pointer of the calling process into the data field ResourceParentA7 of the resource process page. Second, it must load the stack pointer to point to the resource process page data field A5Value. Next, the code must properly initialize address registers A4 and A6 to point to the resource procedure parameters and to the permanent resource variables, respectively. The user stack pointer is also initialized to point directly below the resource procedure parameters. Finally, code is generated to set up an exception stack frame and execute an RTE instruction, forcing the processor into user mode.
After this transition code is created and stored in the resource process page, the current process must be removed from the system queue ReadyQ. In its place, the new process, consisting of the resource process page, is placed onto the system queue ReadyQ. Next, the user mode memory translation for the currently process is removed and replaced with a translation corresponding to the resource process. Memory translation is described in section 5.8 System Memory Translation Management. Finally, the resource is executed by jumping to the code stored in the resource process page.

When the resource procedure call returns, several more tasks must be performed. First, the resource process is removed from the system queue ReadyQ, and calling process is returned in its place. Next, the memory translation for the calling process is restored. The resource procedure parameters are then copied back to the calling program. Finally, after the calling process is finished accessing the resource and its associated memory, the data field ResourceNext2Exec within the resource process page is incremented, allowing other processes access to the resource.

5.7 Host I/O

The BOS operating system interacts with a run time executive that executes on an attached host computer system. The connection between the systems is made by an Ethernet link between the host and the first processor in the parallel processing machine. Ethernet packets sent to the first processor in the parallel
processing machine can be forwarded to other processes by copying the packet into a reserved system global memory page and generating a mailbox interrupt to the receiving processor board. Packets from processors in the parallel processing machine can be sent to the host by copying the packet into the reserved global memory page and generating a mailbox interrupt to the first processor board. The first processor board responds to the mailbox interrupt by copying the packet from the global memory into the Ethernet memory and sending the packet to the host.

All host I/O consists of a series of send packet / receive packet sequences. Each sequence begins with the host computer sending a packet to a specific processor. The receiving processor then responds by sending a response packet back to the host. The first two bytes of each packet sent by the host computer define the manner in which the parallel processing system will respond to the packet. The first byte identifies the processor to which the packet is being sent. For packets sent to the first processor, this byte has a value of one; for packets sent to the second processor, this byte has a value of two, and so on. The second byte of the packet is used to designate the function of the packet. The content of the remainder of the packet is dependent upon the value of the second byte of the packet.

For the context of this discussion, three different types of packets will be discussed. The first packet to be discussed is a packet containing executable code. The second byte of such a packet will have a value of two (2). The next four bytes of this packet represent the long integer address of the code contained within
the packet. The next four bytes of this packet are a long integer value indicating the number of bytes of executable code contained in the packet. The remainder of the packet consists of the actual executable code.

If the byte count value is non-zero, each byte of executable code is stored in the memory of the processor, starting at the specified address. If, on the other hand, the byte count value is zero, the processor responds to the packet by creating a main program process as described in section 5.5 Process Execution. In either case, an acknowledge packet consisting of a single zero (0) byte is then sent to the host computer system in order to verify successful reception of the packet.

The loading and execution of a BPL program therefore proceeds as follows. The host computer breaks the executable code up into packet-sized portions. Each code packet is sent, one at a time, to each processor in the system. Following the sending of each packet, the host waits for an acknowledge packet from the processing node that received the packet. When the entire program is loaded onto all processors, a packet containing the starting address of the program is constructed with a code byte count of zero (0). This packet is sent to the first processor in the system which, after returning an acknowledge to the host, begins executing the main body of the BPL program.

After the BPL program begins execution, the host computer provides file and console input and output services to the parallel processing system. A second type of packets is used to perform these services. A packet in which the second byte has a value of $FD is used to query a processor within the system for I/O service
requests.

When an active process requests an operating system input or output service, that process removes itself from the system queue ReadyQ and joins the system queue EtherQ. The process then relinquishes the processor, enabling the operating system to execute other processes. When a process on the system queue EtherQ is restarted, the processor is servicing an interrupt in response to the reception of a packet from the host computer system. The packet received from the host is in a global receive packet buffer. Additionally, global send packet buffer can be written to by the restarted process. When the restarted process relinquishes the processor, the contents of the global send packet buffer are transmitted back to the host computer system.

Consider the example of a processor attempting to display the character string "Hello World" on the host computer console. The following sequence of events will occur:

1. The process invokes the operating system function for displaying a string. The operating system removes the task from the system queue ReadyQ and places it on the system queue EtherQ. The processor is relinquished to other processes.

2. The host computer system sends a packet to the processor executing the process. This generates an interrupt to the processor.

3. The interrupt handler is invoked. This routine copies the packet into the global receive packet buffer and restarts the process on the system queue EtherQ.

4. The restarted process copies the string "Hello World" into the global send packet buffer. Additionally, it stores other values in the buffer indicating that a string is to be displayed. The process then removes
itself from the system queue EtherQ and joins the system queue ReadyQ. It then relinquishes the processor back to the interrupt handling routine.

5. The interrupt handling routine sends the packet built by the process in the global send packet buffer to the host machine. The interrupt handler returns from the exception, allowing the processor to continue with what it was executing when the interrupt occurred.

6. The host computer system receives the packet. It interprets the contents of the packet and prints the string "Hello World" on the display console.

The operating system responds to the reception of a packet of type $FD by restarting the first process on the system queue EtherQ. If an output operation is performed, the process will copy the data to the global send packet buffer and then return to the system queue ReadyQ. If an input operation is performed, two packets must be received. After receiving the first packet, the process returns a packet to the host indicating the operation to be performed. The process then remains at the beginning of the system queue EtherQ so that the next packet received from the host will cause the same process to be restarted. The host responds to the input request by filling the next packet with the requested data and sending it to the requesting process. The process, when restarted, reads the requested data from the packet. The process then removes itself from the system queue EtherQ and joins the system queue ReadyQ. The process can then continue executing normally.

A large variety of packets can be returned by an executing process to the host computer system. Each of these packets has a unique type value.
Additionally, each packet contains other data required for the requested operation. A special packet also is used to indicate to the host computer system that the specified processor currently has no pending input or output operations. This packet is returned to the host if the system queue EtherQ contains no processes.

The final packet type recognized by the operating system has a value of $FF$. When the processor receives a packet of this type, the currently executing program is aborted and the processor is reset.

5.8 System Memory Translation Management

BPL processes execute in user mode utilizing the virtual memory translation capability of the Motorola 68030 microprocessor. The support of this capability is the responsibility of the operating system.

The 68030 memory management unit (MMU) supports five levels of translation tables. A translation table is an array of pointers. The pointers stored in the last translation table point to physical memory pages. The pointers stored in the higher level translation tables point to lower level translation tables. The address of the highest level translation table is stored in a processor register called the "root pointer." Each level of translation table must have a size that is a power of two. The 68030 has two "transparent translation" registers which can be used to disable address translation for specific address ranges or processor modes.

Memory translation on the 68030 under the BOS operating system works as follows. One transparent translation register is configured so that translation
is disabled when the processor is in supervisor mode. Therefore, the memory translation mechanism is active only when the operating system is executing a user mode process.

The first level translation table contains eight pointers. The 68030 processor produces three function code signals for each memory access. These three bits are used to select one pointer from the first level translation table. The selected pointer points to the appropriate second level translation table. The second level translation table also contains eight pointers. The processor selects one of the eight pointers based upon the value of the three most significant bits of the thirty-two bit address. The selected pointer points to the appropriate third level translation table. The third level translation table is similar in operation to the second level translation table except that it contains sixteen pointers and uses the next four most significant bits of the address to select a pointer to a fourth level translation table. The fourth level translation table contains eight pointers and uses the next three most significant bits of the address to select a pointer to a fifth level translation table.

Finally, the fifth level translation table contains 1024 pointers and uses the next ten most significant bits of the address to select a pointer to a physical memory page. In this manner, the processor translates the upper twenty bits of each logical memory address into a physical memory address. The low order ten bits of each logical memory address are not translated.

One way to provide customized address translation for each BPL process
would be to construct a complete set of translation tables for each active process. In order to conserve memory, however, the BOS operating system allocates only two tables to each process, a fourth level translation table and a fifth level translation table. The fourth level translation is thirty-two bytes in size and, therefore, will fit easily into the process page for the corresponding process. The fifth level translation is four kilobytes in size, requiring the allocation of an entire system memory page. In theory, each of the eight entries in the fourth level translation table could point to a separate fifth level translation table. Currently, however, the BOS operating system restricts each process to one fifth level translation table and, therefore, limits each process to four megabytes of memory.

When a process page is initialized, a fifth level translation table is allocated and initialized to translate memory access by the process to the appropriate physical memory pages. One entry in the fourth level translation table is initialized to point to the fifth level translation table. All other entries in the fourth level translation table are cleared. Any memory access corresponding to one of these other entries will cause an exception to be generated, causing the operating system to halt the program and report the error.

The top three levels of translation tables are the property of the operating system. When a new process is started, the appropriate entry in the third level translation table is set to point to the fourth level translation table for the process. The processor's root pointer register is then reloaded to point to the first level translation table. This causes the processor to reset the memory translation
mechanism.

5.9 BPL Channel Operations

BPL channels, as discussed in section 3.1.3 The Channel, are used for synchronized communication between concurrent processes. The BPL compiler produces special entry code at the beginning of each process that declares a channel variable. This entry code performs a call to the operating system requesting the creation of a new channel. The operating system function allocates a global memory page for the channel and returns the address of the global page. The process then stores this address in the actual channel variable. When the process spawns child processes and passes the channel to the processes as a parameter, the global memory address is the actual parameter value passed to the child.

The BPL compiler also produces special exit code at the conclusion of each process that declares a channel variable. This code calls the operating system in order to release the memory allocated to the channel.

At the beginning of the channel's global memory page, the BOS operating system stores data describing the various operational parameters of the channel. The data structure of this data is as follows:
GChanType = record
  ReadStat,
  WriteStat,
  OpenStat,
  EOCStat,
  ReadProc,
  WriteProc,
  DataSize : longint;
end;

When a new channel is created, the operating system initializes the fields ReadStat, WriteStat, OpenStat, and EOCStat to zero. When any synchronized operation is performed on the channel data structure, the operating system manipulates different bits in each of these fields in order to represent the current state of the channel.

When a reset operation is performed, the value of the data field OpenStat is OR'ed with the value $02$, effectively setting bit number one. Next, the address of the process page for the process executing the reset is stored in the data field ReadProc. Then, the value of the data field OpenStat is OR'ed with the value $20$. At this time, the value of the data field OpenStat is AND'ed with the value $10$. If the result is zero, no corresponding process has executed a rewrite operation on the same channel. In this situation, the process removes itself from the system queue ReadyQ and places itself on the system queue OpenChanQ. If the result is non-zero, a corresponding process has executed a rewrite operation on the same channel. In this case, the current process can continue executing.

The rewrite operation is performed in a similar manner. First, the value of the data field OpenStat is OR'ed with the value $01$. Next, the address of the
process page for the process executing the reset is stored in the data field WriteProc. Then, the value of the data field OpenStat is OR'ed with the value $10. At this time, the value of the data field OpenStat is AND'ed with the value $20. If the result is zero, no corresponding process has executed a reset operation on the same channel. In this situation, the process removes itself from the system queue ReadyQ and places itself on the system queue OpenChanQ. If the result is non-zero, a corresponding process has executed a reset operation on the same channel. In this case, the current process can continue executing.

During the execution of a program, the operating system periodically calls a procedure to check for completed channel reset and rewrite operations. This procedure inspects the system queue OpenChanQ. For each process suspended on this queue, the value of the data field OpenStat for the corresponding channel is examined. If this value AND'ed with $30 produces the value $30, the suspended process can be restarted. This is done by removing the process from the system queue OpenChanQ and placing the process on the system queue ReadyQ.

The operating system performs a channel write in the following manner. First, the data to be written to the channel is copied into the global memory page. The space following the channel data structure is reserved for this purpose. The channel data field WriteStat is then set to a non-zero value. Finally, the process performing the channel write leaves the system queue ReadyQ and joins the system queue SendChanQ, relinquishing the processor to the system.

A read operation begins by checking the value of the channel data field
WriteStat. If this value is zero, no corresponding process has performed a write to the channel. In this situation, the process removes itself from the system queue ReadyQ and joins the system queue RcvChanQ. If the value of the data field WriteStat is non-zero, the process continues. In either case, the next step performed is to move the data from the global memory page into the process variable specified in the read operation. Then, the data field WriteStat is set to zero.

During the execution of a program, the operating system periodically checks for processes suspended on the system queue RcvChanQ. These processes are waiting to read from a channel. For each process found on the queue, the data field WriteStat of the corresponding channel data structure is inspected. If the value of the data field is non-zero, a process has written data to the channel. When this occurs, the process is removed from the system queue RcvChanQ and placed on the system queue ReadyQ. The process is then restarted, allowing it to read the channel data and continue executing.

At the same time, the operating system inspects the system queue SendChanQ searching for processes that have written data to a channel and are waiting for a process to read the written data. For these processes, if the channel data field WriteStat is zero, another process has removed the from the channel. In this situation, the process is removed from the system queue SendChanQ and placed on the system queue ReadyQ, enabling it to continue execution.
Chapter VI

System Demonstration Programs

This chapter describes four demonstration programs written in the BPL language and executed using the BOS operating system implemented on the Pyramid Processing System. These four programs each implement a different model of parallel programming. Each parallel processing construct of the BPL language is used in one or more of these programs, as is each parallel processing operating system function.

The system used to implement these programs consists of six (6) Motorola 68030 microprocessor boards in a VME bus system. Each processor board has a Motorola 68882 numeric coprocessor and one megabyte of local ram. An eight megabyte shared memory board is used by the processors for message and parameter passing.

The timer on the host computer system is used to measure the execution time for each program. This timer is accurate to approximately fifty milliseconds. In order to obtain satisfactory timing measurements, each program contains a sufficient number of iterations to make it at least two orders of magnitude longer than the timer’s resolution.
The run time executive on the host computer system facilitates the timing of programs by inspecting all output to the screen for two string values. The output string ‘^ ^ ^ ^’ is not printed to the screen, but instead causes the timer to be started. The output string ‘&&&&’ is substituted for using the time elapsed since the timer started. These two strings are printed to the console at the beginning and end of each program in order to measure and display the execution time. Each program’s execution time is measured twelve times; the time presented here is the average.

Some of the demonstration programs perform a large amount of disk input and output. For these programs, the timer is started after any input data is read from the disk and stopped before the output data is written to the disk. This is done so that the relatively slow speed of the host computer disk system does not influence the timing.

The programs described in this chapter all perform useful tasks. The programs are not, however, written for maximum efficiency. They are instead written for clarity and to demonstrate a specific aspect of parallel programming using BPL.

Each demonstration program will be presented in the following manner. First the basic algorithm will be discussed. Then, the parallel model for the application is discussed, describing the BPL structures used to implement the model. Finally, the execution time for each program is measured using from one to six processors. Additionally, the timing of an equivalent sequential program
executed on a single processor is measured. These timing results will be presented in tabular form. For each parallel program measurement, the speedup factor is also displayed. This value is the ratio of the execution time using one processor to the execution time for the number of processors considered.

The source code for each demonstration program is contained in Appendix C.

6.1 Edge Preserving Smoothing Program

6.1.1 Algorithm Description

This program implements an image processing algorithm that filters high frequency noise from an image without smearing edge features. The basic algorithm is described in [Nagao, 1979]. The algorithm is implemented as a neighborhood operation, in which each pixel in the resulting output image is calculated based upon the values of a five by five neighborhood of pixels in the original input image.

A traditional smoothing algorithm is performed using a three by three neighborhood averaging operation. This algorithm determines each output pixel by averaging the surrounding nine pixels in the original image. This procedure effectively reduces the amount of high frequency noise in the image. Sharp edges in the original image are also averaged by this process, with the result that the output image appears blurred.

The algorithm used for this demonstration program examines the pixels in
nine different masks within the five by five neighborhood. Eight of these masks correspond to pixels located in the square and diagonal directions relative to the center pixel. The ninth mask corresponds to the same three by three neighborhood used for simple averaging smoothing.

Each of the nine masks are examined in order to locate the one mask which contains the least amount of edge information. This is determined by finding the most homogeneous mask. The variance of each mask is calculated; the mask with the minimum variance is considered to be the most homogeneous. For this mask, the average pixel value is calculated. This average becomes the value of the corresponding output pixel in the resulting image.

For this program, a 200 by 200 pixel image is evaluated.

6.1.2 Parallel Implementation

This program uses a very simple parallel model. Twenty identical processes are utilized, each processing a horizontal "strip" of the original image. No communication takes place between processes. Each process receives one value parameter and returns one return parameter. The input value parameter is a block of the input image. The returned parameter is the resulting block of the output image. Due to the standard property of neighborhood operations, the input image includes two rows above and two rows below the desired output pixel block.

The calculation of the nine different neighborhood values is performed in a structured manner, using a record data type variable to represent the group of
pixels used for the calculation. This data structure is passed to a procedure along with a twenty five element array containing the pixel values. A more efficient approach would be to hard code the equations that extract the neighborhood pixels from the image array. This method, however, results in a more readable program that performs satisfactorily.

The program begins by reading the original image into a single variable large enough to hold the entire image. The execution timer is then started. Next, the main program spawns the pool of twenty processes, sending a portion of the input image to each process. Each process consists of a nested loop that calculates the output values for each row and column of the input image. When the child processes return, the execution timer is stopped and the execution time is displayed. The main program then stores the output image to a file and terminates.

6.1.3 Execution Timing

Table 1 documents the timing of the parallel edge preserving smoothing program using from one to six processors, along with the execution time for a single processor sequential implementation.
Table 1 - Edge Preserving Smoothing Timings

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Execution Time (sec.)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>86.51</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>46.20</td>
<td>1.87</td>
</tr>
<tr>
<td>3</td>
<td>31.80</td>
<td>2.72</td>
</tr>
<tr>
<td>4</td>
<td>26.03</td>
<td>3.32</td>
</tr>
<tr>
<td>5</td>
<td>20.27</td>
<td>4.27</td>
</tr>
<tr>
<td>6</td>
<td>15.97</td>
<td>5.42</td>
</tr>
<tr>
<td>1 Sequential</td>
<td>79.87</td>
<td></td>
</tr>
</tbody>
</table>

6.2 Cumulative Spectral-Decay Calculation Program

6.2.1 Algorithm Description

The program described in this section calculates the data used to produce a three dimensional plot showing the decay of the frequency response of an audio signal. Such a plot plays a role in the design of listening rooms, auditoriums, and loudspeaker systems. [Heyser,1971] describes one method for obtaining the sampled data used to create the plot.

This program processes a sequence of sampled data that corresponds to several seconds of a measured audio signal. The original audio source consists of a short burst of white noise. The signal is measured using a microphone and consists of the original noise burst as modified by the characteristics of the loudspeaker and of the listening room. By performing a frequency response analysis at regular time intervals, the decay characteristics of the system under evaluation can be assessed.

The algorithm employed performs a Fast Fourier Transform (FFT) for each
time interval to be considered. With a stream of input data, this is accomplished by performing multiple overlapping FFT's, windowed in time. The input data is sampled at a fifty kilohertz rate. The time interval for each frequency response analysis is ten milliseconds. By performing a 2048 point FFT, a frequency resolution of approximately twenty-five hertz is obtained. The ten millisecond time interval is achieved by performing an FFT at 500 sample increments. For this example, the data stream consists of 100,000 sixteen bit samples, corresponding to two seconds of sampled audio. Therefore, one hundred ninety-five FFT calculations will be performed.

The desired output is the level of output for each frequency. Each complex element produced by the FFT is reduced to such a value by calculating the magnitude of each frequency component, truncating the result to a sixteen bit integer value.

6.2.2 Parallel Implementation

The parallel program used to implement this algorithm spawns fifteen child processes to perform the FFT calculations. The entire sampled data set is passed to each process as a value parameter. Rather than return the results as a parameter, however, the results of the calculations are stored in a resource. Along with the data set, each process receives a parameter specifying the starting point for the first FFT to be performed by the process. When the process completes the FFT calculation, it calls a resource procedure to store the result.
The starting point of the data used to calculate the FFT is passed with the FFT result to the resource so that the resource can store the combined results in the proper order. The resource then returns to the process the starting point for the next FFT to be performed. When no more transforms are required, the resource returns a value of minus one as an FFT starting point, indicating to the calling process that no more processing is required. The calling process then terminates.

This method of assigning FFT starting points is used in order to distribute the task more evenly among the parallel processes. On processors with fewer processes, the processes will run faster, enabling them to perform the FFT calculation in less time. These processes will therefore call the resource procedure more frequently, requesting further work. In this manner, the processes that are able to run faster due to unbalanced resource distribution are given more work to perform.

The program begins by reading the data set into memory. The execution timer is then started. The main program then calls an initialization procedure to initialize the resource. The child processes are then executed, using a parallel statement. When the parallel statement terminates, the results are obtained from the resource. The execution timer is then stopped and the execution time is displayed. Finally, the main program stores the results in a file and terminates.

The FFT subroutine used in this program is from [Press, 1989].

6.2.3 Execution Timing
Table 2 documents the timing of the cumulative spectral-decay calculation program using from one to six processors, along with the execution time for a single processor executing an equivalent sequential program.

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Execution Time (sec.)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>463.13</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>232.77</td>
<td>1.99</td>
</tr>
<tr>
<td>3</td>
<td>155.28</td>
<td>2.98</td>
</tr>
<tr>
<td>4</td>
<td>117.70</td>
<td>3.93</td>
</tr>
<tr>
<td>5</td>
<td>94.31</td>
<td>4.91</td>
</tr>
<tr>
<td>6</td>
<td>80.52</td>
<td>5.75</td>
</tr>
<tr>
<td>1 Sequential</td>
<td>454.33</td>
<td></td>
</tr>
</tbody>
</table>

### 6.3 Circuit Analysis Program

#### 6.3.1 Algorithm Description

This program is used to calculate the frequency response of an active filter circuit. The circuit contains approximately forty nodes.

The methods described in [Battocletti, 1985] are used to perform the analysis. For a specified frequency, a set of node equations are generated. These equations are transformed into a matrix expression that is reduced using Gaussian elimination. For this example, the circuit will be evaluated at 120 frequencies. These frequencies range from twenty hertz to twenty kilohertz, and are logarithmically spaced.

#### 6.3.2 Parallel Implementation
This program spawns twenty-four child processes. Three parameters are passed to each process. The first parameter is a large data structure that contains a complete description of the circuit. Each element of this parameter describes a component in the circuit, including the type of component, the value of the component, and the nodes to which the component is connected.

The second parameter passed to each process is a resource that the process will use to obtain frequencies for analyzing the circuit, while the third parameter is a resource used to store the results. These resources are used in the same manner as the single resource is used in the cumulative spectral-decay calculation program. They enable the processes which, due to uneven resource distribution, are executing faster to perform more of the work.

The first action taken by the main program is to start the execution timer. The program then constructs the data structure that describes the circuit. The circuit created is an active filter circuit with selected gains at specified frequencies. Several procedures are provided to allow the programmer to specify a variety of gains at these frequencies, inserting components into the circuit according to the parameters passed to the procedures.

The main program then calls a resource initialization procedure to initialize the resource that will provide frequencies to each of the child processes. This initialization establishes the first frequency to be used and the ratio between successive frequencies.

The child processes are then executed, using a parallel statement. When the
parallel statement terminates, the main program stops the execution timer and displays the execution time. A resource procedure is then called to display the results in order of increasing frequency.

6.3.3 Execution Timing

Table 3 documents the timing of the circuit analysis program using from one to six processors, along with the execution time for a single processor sequential implementation.

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Execution Time (sec.)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>129.68</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>78.16</td>
<td>1.66</td>
</tr>
<tr>
<td>3</td>
<td>53.05</td>
<td>2.44</td>
</tr>
<tr>
<td>4</td>
<td>38.17</td>
<td>3.40</td>
</tr>
<tr>
<td>5</td>
<td>31.47</td>
<td>4.12</td>
</tr>
<tr>
<td>6</td>
<td>24.55</td>
<td>5.28</td>
</tr>
<tr>
<td>1 Sequential</td>
<td>122.43</td>
<td></td>
</tr>
</tbody>
</table>

6.4 Image Compression Program

6.4.1 Algorithm Description

This demonstration program uses a three-step operation to compress a raster scanned image, introducing some image degradation. The amount of degradation is specified by the user.

The image to be compressed is divided into eight by eight neighborhoods. Each neighborhood is processed independently; the results of the processing are
concatenated and stored in an output file.

The first step is to perform a two dimensional discrete cosine transform (DCT) on the neighborhood. This operation is discussed in [Press,1989]. The result of the DCT is an eight by eight matrix of spectral values. The position of each element in the matrix corresponds to the spectral component the element represents. Element (0,0) represents the DC component of the neighborhood; for most neighborhood, this value is much larger than all other elements. Elements which are located further from element (0,0) represent higher frequency components of the input neighborhood; these elements are typically very small.

The second step of the compression is to quantize the values of the resulting DCT matrix. The elements corresponding to low frequency components of the neighborhood are quantized with higher resolution than the elements that correspond to high frequency components. The actual quantization values used are a function of the acceptable degradation as specified by the user.

The final step of the compression is to encode the output. The elements of the quantized DCT matrix are written out in a zig-zag fashion in order to approximately group the elements in order of increasing frequency content. In doing so, successive zero values are frequently encountered. One technique utilized in the output encoding is to run-length encode zero values. Other methods include Huffman encoding of the output values and variable length bit sequences.

6.4.2 Parallel Implementation
The program used here is a derivation of a program described in [Nelson, 1991]. This program differs in the method used for encoding the output values. Each output value is encoded as a sequence of one or more four-bit nibbles. The last nibble corresponding to a value has its most significant bit set. Preceding nibbles have this bit cleared. Each nibble therefore represents three bits of the output value. An output value of zero is immediately followed by a second number indicating the number of successive zeroes. For example, a string of five zeroes would be represented by a zero value followed by a value of five.

The nibble encoding scheme is used in order to facilitate the concatenation of blocks of data from multiple concurrent processes. A simple scheme is used for blocks that contain an odd number of nibbles. Three nibbles are added to the end of such a block. These three nibbles correspond to two zero values. The decoder would interpret these values as a zero-length string of zero values.

Three types of processes are used for this program. The first process is used to send "strips" of pixels to other processes. A select statement is used to select a ready process to receive the pixels on a channel. This process is then sent a strip of pixels, along with an integer value representing the strip number. When no further strips remain to be processed, a message is sent with a strip number of negative one. The receiving process uses this value as an indication that it should terminate.

The second process is used to receive the blocks of data corresponding to a compressed pixel strip. The process uses a select statement to select a ready
process that is writing data to a channel. The sending process sends two other values along with the data, indicating the pixel strip number and the size of the data block.

The final process is used to perform the actual compression. This process first reads a strip of pixels from an input channel. The compression is then performed, with the results written to an output channel. When no further strips of pixels remain to be processed, the process terminates. Ten processes of this type are created.

The program begins by reading the image from a disk file. The execution timer is then started. The main program then executes a parallel statement. The ten compression processes are each sent a single input channel and a single output channel. The process that supplies strips of pixels is sent two parameters. The first parameter is an array of ten channels, one for each compression processes. The second parameter is the image data. Finally, the process that receives the compressed data receives a single parameter, an array of ten channels.

The process that receives the compressed data counts the number of strips it has received. When all strips are received, it concatenates the individual results and writes the data to a disk file.

6.4.3 Execution Timing

Table 4 documents the timing of the image compression program using from one to six processors, along with the execution time for a single processor
sequential implementation.

Table 4 - Image Compression Program Timing

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Execution Time (sec.)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59.05</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>33.83</td>
<td>1.75</td>
</tr>
<tr>
<td>3</td>
<td>25.65</td>
<td>2.30</td>
</tr>
<tr>
<td>4</td>
<td>21.09</td>
<td>2.80</td>
</tr>
<tr>
<td>5</td>
<td>17.69</td>
<td>3.34</td>
</tr>
<tr>
<td>6</td>
<td>16.09</td>
<td>3.67</td>
</tr>
<tr>
<td>1 Sequential</td>
<td>48.88</td>
<td></td>
</tr>
</tbody>
</table>

6.5 Remarks

The programs presented here each demonstrate acceptable speedup factors. One reason for this is that the programs all utilize coarse grain parallelism; each process performs a large amount of processing before synchronizing with other processes.

In general, processes that perform little or no synchronization with other processes will execute most efficiently. Many applications, however, cannot be composed of processes that execute independently. Other applications can be made more efficient by periodic synchronization. An example of this type of application is a "divide and conquer" program where each portion of the complete task requires a non-uniform amount of processing. In order to perform such an application using independent processes, each process would be assigned a portion of the overall task. Processes with portions that require little processing would
terminate quickly, while those with more complex processing requirements would require more processing time. The result would be a program that does not distribute the work load evenly between concurrent processes.

The alternative to independent processes for such an application is a group of processes that communicate periodically. Each communication would consist of a request for more work. Processes that receive data corresponding to little processing will quickly process the data and request more work. Processes receiving data requiring more execution time would execute longer before requesting more work.

The program that demonstrated the worst speedup figures is the image compression program. This program differs from the others in that channels, in conjunction with select statements, are used to distribute the task. The performance of this program is limited by the processes that execute the select statements; when these processes are temporarily blocked by the operating system, all other processes in the system attempting to communicate are also blocked. One solution to this predicament would be to assign a higher priority to processes executing select statements.

The efficiency of programs which utilize resources also can suffer when processes are blocked by the operating system. Consider the situation where a process exits a resource, allowing another process access to the resource. If that next process is blocked, all other processes waiting for access to the resource will be blocked. One solution to this problem would be to prioritize all processes
executing within resources.

Both of these problems can be reduced by decreasing the interval between time slices. This decrease, however, introduces other inefficiencies since the operating system spends more time switching processes and, therefore, less time executing processes. The time slice interval of 50 milliseconds is found to represent a suitable compromise.
Chapter VII
Conclusion

The basic goals of this research project were successfully achieved. A programming language that facilitates the coding of parallel application programs is specified. A reliable compiler is written for the language. An operating system is written that efficiently executes the resulting programs.

This programming system allows parallel programs to be written in a high level language, using new data types and control structures to express relationships between concurrent processes. The operating system performs the job of distributing processes onto processors, relieving the programmer of this task. Sample application programs demonstrated good speedup ratios for different system configurations.

The BOS operating system is designed to be transportable to different machines. This goal proved to be desirable, as the initial Pyramid Processing System became inoperable, forcing a new system to be constructed. This new system consists primarily of commercially available processor and memory components. Porting the operating system to the new machine required modifications to the memory address translation service routines, to the inter-
processor communication service routines, and to the host computer interface routines. The operating system was ported to the new system in less than three weeks.

Finally, the programming system enables experimentation on various configurations of the Pyramid Processing System. The language is based on the Pascal language, facilitating the learning of the new language. The project described in [Nikunie, 1992] used the BPL language and the BOS operating system extensively to test the hardware and to implement several test programs.

This dissertation consists primarily of three main tasks: the specification of the language, the creation of a compiler for the language, and the writing of an operating system to support the runtime execution of programs written in the language. Further work to performed related to each of these tasks will now be discussed. A summary presents the contribution of the work.

7.1 Further Language Definition Work

The existing BPL language structure provides a useful tool for experimenting in parallel programming. During the development of the demonstration programs, it became clear that a reliable mechanism for assigning tasks dynamically in a "divide and conquer" fashion is critical for efficient program execution. This mechanism is provided here through the use of resources and channels. Both of these methods, however, require the programmer to create the code that assigns tasks and notifies the processes when no more work remains. This code is
functionally replicated in each of the demonstration programs.

An obvious improvement would be to shift this functionality away from the application programmer and implement it instead as a language structure. One form of this approach is used in the Linda programming system, described in [Gelernter, 1985] and [Gelernter, 1988]. The Linda system is composed of concurrent processes. These processes search for and extract tasks from a Tuple Space (TS), a construct implemented by the system. When tasks are completed, the results are returned to the TS. Further tasks are then removed from the TS, until no more tasks remain.

This type of functionality could be incorporated into BPL through the addition of a new data type and a new control structure. The data type would be an abstract type, allowing the specification of tasks. The control structure would spawn multiple processes to respond to the tasks represented by the abstract type.

Other types of functionality can be added to BPL. One obvious need is the inclusion of a queuing mechanism in resources. Such a mechanism is utilized in [Brinch Hansen, 1977]. This mechanism allows processes executing within a resource to join a queue if needed data is not present. Subsequent processes, executing within the same resource, can remove the processes from the queue by providing the required data.

The channel of BPL is a synchronous mechanism. A likely extension to the channel would be the creation of non-blocking channels. Another added facility would be a function for testing the readiness of a channel so that a read or write
could be performed without blocking.

The current BPL language is based on the sequential language Pascal. The C programming is more popular than Pascal, but does not provide the strict type checking facilities that Pascal requires. The C++ language, on the other hand, is a superset of C that implements strict type checking. C++ also provides other facilities which could be utilized for parallel programming.

C++ is designed so that the language can be extended with new data types. Furthermore, the existing language operators can be defined to provide new functionality when used with new data types. One possibility would be to develop an object type representing a concurrent process. The addition operator could be defined for such an object type to invoke the execution of multiple processes simultaneously. Other object types and operators could be defined to provide inter-process communication or access to critical code regions.

One benefit of the C++ language is that it is commonly available on a broad variety of platforms. This large installed base could be used for the development of experimental parallel programs. Furthermore, network workstations equipped with C++ compilers could execute concurrent programs.

### 7.2 Further Compiler Work

The existing BPL compiler, while quite functional and reliable, does not produce highly efficient code. The only optimizations utilized are a few localized "peep hole" techniques. Several interesting approaches to achieving higher
executable code efficiency are described in [Milewski, 1985] and [Paduao, 1986]. The structure of the existing BPL compiler is flexible enough to allow the incorporation of a variety of different optimization schemes.

The BPL compiler is currently written to generate code for the Motorola 680x0 microprocessor. A useful exercise will be to write a new code generation module so that other processors can be supported. In particular, the Intel 80386 processor is a good candidate for code generation. This processor is currently available in very low cost systems. A parallel language that supports the 80386 could be used to construct a low cost parallel processing machine.

The compiler front end supports the translation of BPL programs, which are very similar to Pascal. In order to develop a new language based on C++, this front end will need to be rewritten. This task, in combination with the rewriting of the code generator, represents a major revision of the compiler. In order to support future revisions as well, the compiler should be restructured so that the front end and the code generator can be changed independently. Such a structure would facilitate the incorporation of new language features or the support of new processor architectures.

The restructured compiler would be designed so that a substantial portion of the optimization processing is performed by an intermediate program module. This optimization module would be independent of the source language and the target architecture.

The development of efficient parallel processing application programs is a
difficult task. In order to better facilitate this work, a program debugger and an execution profiler should be implemented. The compiler will facilitate this by constructing tables of information during program compilation. These tables will include the address of each program statement, the names of all variables for each scope level of the program, and the location within the program of each process synchronization operation.

The debugger will load the program and place break points at the addresses corresponding to user-specified program statements. When the program stops at break points, the user will be able to inspect the values of all variables in the current program scope. The user also will be able to single step through the program while the debugger displays the next statement to be executed.

The execution profiler will be used to measure a variety of program execution parameters. These parameters include the number of iterations for a specified loop, the number of times a selected statement is executed, the number of times a specific channel is accessed by a pair of processes, the execution time for a specified process, or the total number of child processes spawned by a specified parent. These parameters can be useful when fine-tuning the performance of a program.

7.3 Further Operating System Work

The BOS operating system is highly experimental in nature, and is therefore a good candidate for further work. The operating system routines which perform
process mapping and task scheduling are good candidates for further experimentation. Additionally, planned changes to the system hardware will require operating system support.

[Nikuie, 1992] describes a point-to-point communication network designed for the Pyramid Processing System. The operating system will be modified to make use of this network. It is not yet clear what portions of the operating system will use the network.

Inter-processor communication is used to implement channels, resources, and concurrent process execution. Each of these entities are currently implemented using shared memory. Resources are implemented very efficiently using shared memory due to the fact that a resource must maintain a local variable space. Channels, on the other hand, are currently implemented rather inefficiently using shared memory. This is due to the fact that writing to the shared memory is not immediately detected by another processor. The communication network, however, generates an interrupt when data is received, providing a more efficient means of implementation. Channels therefore are a good candidate for using the communication network.

An efficient network implementation of process execution might involve a hybrid of communication network and shared memory techniques. The fast notification time of channels would permit prompt execution of newly created processes. The large amount of parameter data passed to and from processes can more easily be transferred using shared memory.
Section 7.2 Further Compiler Work discusses compiler enhancements needed to implement a parallel program debugger and an execution profiler. The operating system will also require modifications in order to support these tools. The debugger and the profiler will be implemented as application programs. These programs will load the program to be evaluated into the system and control the program's execution. A variety of operating system functions must therefore be made available to application programs in order to facilitate the development of the debugger and the profiler.

These new functions will provide the ability for an application program to load another program, place break points within the other program, restart other programs that have reached break points, and examine and modify variables within the private memory space of the other program. New system queues will be implemented, enabling debugger or profiler programs to be blocked until the program under inspection reaches a breakpoint. More advanced timer capabilities will be incorporated in order to measure process execution times accurately. Special "hooks" will be placed into the operating system, enabling a debugger or profiler to be signaled, for example, when child processes are spawned or when two processes communicate through a channel.

The currently implemented host user interface is rather primitive. User input and output operations are performed using a terminal style interface. Because the Pyramid Processing System is intended primarily for image processing research, a graphical user interface is desirable. This interface should allow images
to be displayed. Additionally, the user should be able to draw and select image areas using a pointing device such as a mouse. These enhancements will require the addition of several user interface operating system functions.

Because of the many changes to be made to the BOS operating system, a different structure should be implemented. Currently, all operating system functions are contained in a single program which is programmed into EPROM memory on each processor board. This approach does not readily facilitate operating system modifications. A better structure would be a micro-kernel operating system. In such a system, the operating system is formed around a very small kernel. This kernel merely serves to link the various components of the operating system. Each component of the operating system is then loaded independently and linked dynamically to the kernel. These components can be modified and linked quickly, allowing faster operating system testing and experimentation.

7.4 Summary

The result of this dissertation project is a system that incorporates several concepts unique to parallel processing programming. The language is structured such that it is applicable for a broad class of MIMD machines rather than designed for a specific architecture. Resulting programs are written for a virtual parallel processing machine rather than tailored to a specific one. Using this system, parallel processing programs are expressed in a high level language that reflects
the parallel structure of the resulting program.

The operating system is novel in that it automates the task of machine allocation in a manner that is dependent upon inter-process communication requirements. This is in part facilitated by the "hints" provided to the operating system by the BPL compiler. The close interaction between the compiler and the operating system is unique, as is the use of process communication requirements when performing the machine allocation.

[Austin, 1991] discusses a project that shares many attributes of this dissertation. The paper was published following the completion of this work. The subject of the paper is the Wisdom Project, a parallel processing system consisting of Transputer microprocessors under the control of a distributed operating system. Programs are written for the system using the Occam language.

In a manner similar to the BPL language and BOS operating system, the Wisdom Project system presents the programmer with a scalable system of "virtual" processors which have "location transparency." Additionally, an attempt is made to allocate processors to processes based upon inter-process communication requirements. Austin states that this is perhaps the first attempt to do this. The operating system assumes that parent and child processes will attempt to communicate with one another, and therefore places child processes near the parent throughout the distributed processor network.

The BPL/BOS system goes further than the Wisdom Project in that the compiler presents "hints" to the operating system regarding the communication
relationships of concurrent processes. Additionally, the BPL language also incorporates resources, for which there is no counterpart in Occam. The BOS operating system also considers accesses to resources when performing the machine allocation. Therefore, this dissertation represents a contribution beyond that of Austin.

It is certain that the parallel programming languages of the future will be very different from this project and other existing parallel programming systems. Current systems bear a strong resemblance to standard sequential programming languages. In fact, programs written in the language created for this dissertation consist of concurrent processes that are themselves sequential modules. New developments in parallel programming languages will likely move away from such strong connections to conventional sequential programming structures. The result will be programming techniques that will further facilitate the expression of parallel algorithms.
Bibliography


A PROGRAMMING LANGUAGE AND OPERATING SYSTEM FOR PARALLEL PROCESSING COMPUTING MACHINES

VOLUME II

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Gregory John Bergmann, B.S.A.R., M.S.

* * * * *

The Ohio State University

1992

Dissertation Committee:

J. M. Jagadeesh

H. Hemami

S. Ahalt

Approved by

J. M. Jagadeesh

Advisor

Department of Electrical Engineering
# TABLE OF CONTENTS

## APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>BPL Compiler Source Code Listing</td>
<td>359</td>
</tr>
<tr>
<td>B.</td>
<td>BOS Operating System Source Code Listing</td>
<td>594</td>
</tr>
<tr>
<td>C.</td>
<td>Demonstration Programs Source Code Listings</td>
<td>669</td>
</tr>
</tbody>
</table>
Appendix A

BPL Compiler Source Code Listing

{$A-,B-,D+,E+,F+,I+,L+,N+,O+,R+,S+,V+}$
{$M 32768,0,655360}$

program compile;

uses crt,
    dos,
    sprfivar, {scans procedures, functions, VAR declarations}
    scontyp, {used to scan constant and type declarations}
    sstat, {used to scan all statements}
    labels, {used to generate internal labels and storage areas}
    lblock, {keeps track of current block level}
    lex, {reads in files, converts to lex tokens}
    getsym, {used to obtain next token}
    types, {declares all types used}
    strstore, {used to store and retrieve string constants}
    options, {used to flag all options - call to check options}
    tables, {stores all compiler created tables}
    gencode,CodeEmit, {used to generate pseudocode}
    list, {used to generate listing - for develop only}
    CircRef, {used for procedural variables}
    scparms,
    scanpar,
    errors; {used to flag all errors}

var
    FName : DataString;
    row,col : integer;
    fnumber : integer;
    nested : boolean;
    StartTime,
    EndTime : record
        Hour,
        Minute,
        Second,
        Sec100 : word;
        Sec100SinceMid : longint;
    end;
    TotalTime : longint;
ErrFile : text;
ErrLine : string[79];

i:integer; jj:string[20];

procedure ScanProgDec;
{ this procedure scans the program declaration }
{ it does not, however, do anything with the declaration }
begin
  get; {clear out the PROGRAM}
  if LexSym.token=identifier1 then get
  else FlagError(2);
  if LexSym.token=LeftParl then begin
    get; {clear out the '('}
    if LexSym.token=identifier1 then get
    else FlagError(2);
    while LexSym.token=commal do begin
      get; {clear out the comma}
      if LexSym.token=identifier1 then get
      else FlagError(2);
    end;
    if LexSym.token=RightParl then get
    else FlagError(4);
  end;
  expect(semicolon1,14);
end;

procedure ScanUnitDec;
begin
  get; {unit}
  if LexSym.token <> identifier1 then FlagError(2);
  get;
  expect(semicolon1,14);
end;

procedure ScanInterface;
begin
end;

procedure ScanImplementation;
begin
end;

procedure ScanBody;
begin
  ScanDeclarations;
  if LexSym.token <> beginl then FlagError(17);
  Emit(ProgStart,VarAddrs[0]);
  Emit(InitPF,0);
  NewExitLevelNewLabel);
  ScanCmpdStat;
Emit(DecLabel, ExitLabel);
ForgetLabel(ExitLabel);
EndExitLevel;
Emit(ClosePF, 0);
end;

procedure ScanProgram;
begin
Emit(InitStr, 0);
if (LexSym.token = program) then ScanProgDec
else if LexSym.token = unit then begin
ThisIsUnit := true;
ScanUnitDec;
InsideImplementation := false;
ScanInterface;
InsideImplementation := true;
ScanImplementation;
end;
if not ThisIsUnit then begin
InsideImplementation := false;
Emit(JmpAlways, 0);
ScanBody;
Emit(ProgEnd, 0);
Emit(CloseStr, 0);
{ InsertFPCode; }
Emit(SourceCodeEnd, 0);
if LexSym.token <> period then FlagError(23);
DumpTabels;
EndBlock;
end;
end;

{$F+}
procedure DoCompilation(t:longint);
begin
{Save stack pointer here - it will be used to return to the main program }
{directly from the error routine in the event of a compilation error}
AbortAddress := addr(t);
ScanProgram;
end;

begin
writeln('BPL Compiler, Version 7.95');
if ParamCount = 0 then begin
writeln('Syntax :');
writeln('COMPILE <file name>');
writeln;
writeln('No File Extension is Necessary - A Default Extension of .BPL is Used.');
halt;
end;
with StartTime do begin
  GetTime(Hour,Minute,Second,Sec100);
  Sec100SinceMid := Sec100 + 100 * (Second + 60 * (Minute + 60 * Hour));
end;
InitList;
error := false;
BlockLevel := 0;
VarAddrs[0] := 0;
InitLexTables;  {unit lex}
InitStringStorage;  {unit strstore}
InitOptions;  {unit options}
InitTables;  {unit tables}
ThisIsUnit := false;
GencodeInit;
InitLabels;  {unit labels}
if (ParamStr(2) = '/e') or (ParamStr(3) = '/e') then LocateErrorSetup;
FName := ParamStr(1);
MainFileName := FName;
if posC',MainFileName) <> 0 then
  delete(MainFileName,posC',MainFileName),255);
AssignInputFile(FName);
get;
DoCompilation($04030201);
writeln;
CloseList;
if error then
begin
  GetStatus(row,col,fnumber);
  if ErrorLocated then
    begin
      row := ErrorRow;
      col := ErrorCol;
      fnumber := ErrorFnum;
    end;
  GetFileName(FName,fnumber);
  assign(ErrFile,FName);
  writeln('Error File : ',FName);
  reset(ErrFile);
  writeln('*** ERROR LOCATED ***');
  writeln('Approximate Error Location :');
  while row>0 do
    begin
      readln(ErrFile,ErrLine);
      if row <= 5 then writeln(ErrLine);
      dec(row);
    end;
  close(ErrFile);
gotoxy(col,WhereY);
  writeln('^');
writeln(ErrorMessage);
GetStatus(row,col,fnumber);
if ErrorLocated then
begin
  row := ErrorRow;
  col := ErrorCol;
  fnumber := ErrorFnum;
end;
writeln('Error in File : ',FName);
writeln('Error at Line : ',row);
halt(5);
end;
DiscardStringStorage;
DiscardLexTables;
DiscardLabels;
DiscardTables;
with EndTime do begin
  GetTime(Hour,Minute,Second,Sec100);
  Sec100SinceMid := Sec100 + 100 * (Second + 60 * (Minute + 60 * Hour));
  if Sec100SinceMid < StartTime.Sec100SinceMid then
    Sec100SinceMid := Sec100SinceMid + 100 * 60 * 60 * 24;
  TotalTime := Sec100SinceMid - StartTime.Sec100SinceMid;
end;
write(Lex.TotalLines,' lines, ');
write(TotalTime/100:0:1,' seconds, ');
write(Labels.CurrentAddress,' bytes code, ');
writeln(VarAddrs[0],' bytes data. ');
end.

unit scanpar;
{used to scan process & resource declarations}
interface
uses crt,
    lex, {used to obtain input tokens}
    labels, {used to generate internal labels and storage areas}
    lblock, {keeps track of current block level}
    getsym, {used to obtain next token}
    types, {declares all types used}
    options, {used to flag all options - call to check options}
    tables, {stores all compiler created tables}
    gencode,codeemit, {used to generate psuedo code}
    CircRef, {used for procedural variables}
    errors; {used to flag all errors}

procedure ScanProcessTypeAct(var tptr:integer);
procedure ScanResourceTypeAct(var tptr:integer);
procedure ScanParallelStatAct;
procedure ScanProcessCallAct;

implementation

procedure EmitTypeDesc(TypPtr : integer);
var
i : word;
begin
if not ContainsType(ChannelType,TypPtr,0) AND
    not ContainsType(ResourceType,TypPtr,0) then begin
    Emit(DecWord,0);
    Exit;
end;
case TypeTabl^[TypPtr].TypeDesc of
    ChannelType : Emit(DecWord,1);
    ResourceType : Emit(DecWord,2);
    ArrayType : begin
        Emit(DecWord,9); { array indicator }
        with TypeTabl^[TypPtr] do begin
            with TypeTabl^[P1] do
                Emit(DecLong,P2 - P1 + 1); { number of elements }
            Emit(DecLong,TypeTabl^[P2].TypeSize); { stride }
            EmitTypeDesc(P2);
        end;
    end;
    RecordType : begin
        Emit(DecWord,10); { record indicator }
        Emit(DecWord,TypeTabl^[TypPtr].P2); { # of fields }
        with TypeTabl^[TypPtr] do for i := P1 to P1 + P2 - 1 do with FieldTabl^[i] do begin
            Emit(DecLong,AddrOffs);
            EmitTypeDesc(FTypEPtr);
        end;
    end;
else Emit(DecWord,0);
end;
end;

procedure ScanProcessTypeAct(var tptr:integer);
var ProcessLabel,ParmSize,
    FirstParm,NumParm,StartLabel,i,TypSize,PTypSize : longint;
    MemSize : longint;
begin
    get; {PROCESS}
    ProcessLabel :=NewLabel;
    FirstParm := ParamTablPtr + 1;
    StoreTypeData(ProcessType,ProcessLabel,FirstParm,0,12,3);
    tptr := TypeTablPtr;
    StartNewBlock;
    StartProcessLevel;
    if LexSym.token=LeftPar1 then ScanParmDec(ParmSize,true);
    NumParm := ParamTablPtr - FirstParm + 1;
    TypeTabl^[tptr].p2 := NumParm;
    { Now store the size of the process variable:
      1 word for status,
      1 long word for link address of next process variable,
      1 long word for code address,
PTypSize := 12;
{add in appropriate size for each parameter}
for i := FirstParm+NumParm-1 downto FirstParm do begin
  case ParamTabl ^ [i].IsVar of
    ValPType : if TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeDesc in
      [RecordType,ArrayType,ChannelType,SharedType,
      ResourceType] then inc(PTypSize,4)
    else PTypSize := PTypSize +
      TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeSize;
  VarPType, RetPType : inc(PTypSize,4);
  end;
  PTypSize := (PTypSize + 1) and $fffffffe;
end;
TypeTabl ^ [tptr].TypeSize := PTypSize;
expect(semicolon,14);
ScanDeclarations;
if LexSym.token <> begin1 then FlagError(17);
Emit(Initialize,0);
Emit(DecLabel,ProcessLabel);
StartLabel := NewLabel;
Emit(JmpAlways,StartLabel);
MemSize := DefaultStack + VarAddrs[BlockLevel];
Emit(DecLong,MemSize);
Emit(DecWord,NumParm);
for i := FirstParm+NumParm-1 downto FirstParm do begin
  case ParamTabl ^ [i].IsVar of
    ValPType :
      case TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeDesc of
        RecordType : Emit(DecWord,$0001);
        ArrayType : Emit(DecWord,$0002);
        ChannelType : Emit(DecWord,$0011);
        SharedType : Emit(DecWord,$0201);
        ResourceType : Emit(DecWord,$0301);
        else Emit(DecWord,$0000);
      end;
    VarPType : Emit(DecWord,$0002);
    RetPType : Emit(DecWord,$0003);
  end;
  TypSize := TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeSize;
  Emit(DecLong,TypSize);
end;
for i := FirstParm+NumParm-1 downto FirstParm do
  if (ParamTabl ^ [i].IsVar = ValPType)
    AND (TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeDesc in
      [RecordType,ArrayType])
  then EmitTypeDesc(ParamTabl ^ [i].TypePointer);
Emit(declabel,StartLabel);
ForgetLabel(StartLabel);
Emit(StartProcess,VarAddrs[BlockLevel]);
if LexSym.token <> begin then FlagError(17);
Emit(InitPF,0);
ScanCmpdStat;
Emit(ClosePF,0);
expect(semicolon,14);
Emit(GetLabelAddress,ProcessLabel);
Emit(EndProcess,0);
EndProcessLevel;
EndBlock;
expect(end,13);
end;

procedure ScanResourceTypeAct(var tptr:integer);
var row,col,fhum,iprocl:integer;
FirstProc,NumProc,FirstParm,NumParm,i,ParmSize:longint;
begin
get; {RESOURCE}
expect(interface,62);
FirstProc := FieldTablPtr + 1;
StoreTypeData(ResourceType,DefaultStack,FirstProc,0,4,1);
 Commercial
expect(implementation1,63);
ScanDeclarations;
expect(end1,13);
TypeTabl[tpr].VariableSpace := VarAddr[BlockLevel];
EndProcessLevel;
EndBlock;
end;

procedure ScanProcessCallAct;
var PType:integer;
begin
  ScanVarAddress(PType,{HasFuncCall},false,false);
  if TypeTabl[PType].TypeDesc <> ProcessType then FlagError(289);
  Emit(PushVar,0);
  with TypeTabl[PType] do 
  ScanParams(p,0,{false},true);
  {push the process address onto the stack}
  EmitCGetLabelAddress,TypeTabl[PType].SubType);
  Emit(PushVar,0);
  Emit(CopyParms,TypeTabl[PType].TypeSize);
end;

procedure ScanParallelStatAct;
var ChType : integer;
ThisStat,
NextEvalLabel,
EndLabel : longint;
begin
  case LexSym.token of
  parallel1 : begin
    get;
    {*** Push a longword=-1 to indicate the last process ***}
    Emit(IntConstant,-1);
    Emit(PushVal,LongintTypePtr);
    if not (LexSym.token in [identifier1.for1]) then 
    FlagError(2);
    while LexSym.token in [identifier1,fori] do begin 
    case LexSym.token of
    identifier1 : ScanProcessCall;
    fori : ScanForStat(true);
    else FlagError(-5);
      end;
      if LexSym.token <> end1 then expect(semicolon1,14);
    end;
    expect(end1,13);
    Emit(ParallelCall,0);
    end;
  select1 : begin
    get; {select}
    EndLabel := NewLabel;
    }
Emit(IntConstant,-1);
Emit(PushVal,LongintTypePtr);
while not (LexSym.token in [else1,end1]) do begin
  NextEvalLabel := NewLabel;
  ScanVarAddress(ChType,{HasFuncCall,false,false);
  if TypeTabl [ChType].TypeDesc <> ChannelType then
    FlagError(226);
  expect(colon1,5);
  ThisStat := NewLabel;
  Emit(ChkSelect,ThisStat);
  Emit(JmpAlways,NextEvalLabel);
  Emit(declabel,ThisStat);
  ForgetLabel(ThisStat);
  ScanStat;
  if LexSym.token <> end1 then expect(semicolon1,14);
  Emit(JmpAlways,EndLabel);
  Emit(declabel,NextEvalLabel);
  ForgetLabel(NextEvalLabel);
end;
if LexSym.token = else1 then begin
  NextEvalLabel := NewLabel;
  Emit(SkipElse,NextEvalLabel);
  get;
  ScanStat;
  Emit(JmpAlways,EndLabel);
  if LexSym.token=semicolon1 then get;
  Emit(declabel,NextEvalLabel);
  ForgetLabel(NextEvalLabel);
end else Emit(PushLabelAddr,EndLabel);
expect(semicolon1,13);
Emit(ChooseSelect,0);
Emit(declabel,EndLabel); ForgetLabel(EndLabel);
end;
else FlagError(-5);
end;
end;

begin
  ScanProcessType := ScanProcessTypeAct;
  ScanResourceType := ScanResourceTypeAct;
  ScanParallelStat := ScanParallelStatAct;
  ScanProcessCall := ScanProcessCallAct;
end.
unit scparms;

interface
uses
crt,
getsym, {used to obtain next token}
lex, {used to get status reports}
lblock, {keeps track of block level}
types, {declares all types used}
options, {used to flag all options - call to check options}
tables, {stores all compiler created tables}
CircRef, {used for procedural variables}
errors; {used to flag all errors}

implementation

procedure ScanParmDecAct(var TotParamSize:longint;IsPar:boolean);
const MaxNumIds = 50;
var IdNums : array[1..MaxNumIds] of integer;
i,count : integer;PTypePtr : integer;
AddrOffset : longint;
row,col,fnum:integer;
TotalCount : integer;
CurrentType : ParmType;
begin
get; {clear out the ( }
AddrOffset := 0;
TotalCount := 0;
if not (LexSym.token in \[ret1,var1,identifier1]]) then FlagError(2);
while (LexSym.token in \[ret1,var1,identifier1]) do begin
  count := 1;
case LexSym.token of
    identifier1 : CurrentType := ValPtype;
    var1 : begin get; CurrentType := VarPtype; end;
    ret1 : begin
      if not IsPar then FlagError(288);
      get; CurrentType := RetPtype;
    end;
end;
if LexSym.token <> identifier1 then FlagError(2);
if AlreadyDef(LexSym.i1) then FlagError(101);
IdNums[1] := LexSym.i1;
inc(TotalCount);
get;
while (LexSym.token= comma1) do begin
  get;
  if LexSym.token <> identifier1 then FlagError(2);
  if AlreadyDef(LexSym.i1) then FlagError(101);
  for i := count downto 1 do
    if IdNums[i]=LexSym.i1 then FlagError(101);
  inc(count);
  inc(TotalCount);
  if count>MaxNumIds then begin
    dec(count);
    FlagError(266);
  end;
  IdNums[count] := LexSym.i1;
  get;
end;
expect (colon, 5);
GetStatus (row, col, fnum);
ScanType (PTypePtr);
if TypeTabl ^ [PTypePtr].level = BlockLevel then
  FlagErrorPos (191, row, col, fnum);
if TypeTabl ^ [PTypePtr].TypeDesc = FwdPtrType then
  FlagErrorPos (118, row, col, fnum);
{######## Add checks here for illegal parameter types. ########}
if IsPar then begin
  if ContainsType (FileType, PTypePtr, 0) then
    FlagErrorPos (305, row, col, fnum);
  if ContainsType (TextType, PTypePtr, 0) then
    FlagErrorPos (305, row, col, fnum);
  if ContainsType (ProcessType, PTypePtr, 0) then
    FlagErrorPos (306, row, col, fnum);
  if ContainsType (PtrType, PTypePtr, 0) then
    FlagErrorPos (317, row, col, fnum);
  if ContainsType (ChannelType, PTypePtr, 0) and
     (CurrentType <> ValPtype) then
    FlagErrorPos (307, row, col, fnum);
  if ContainsType (SharedType, PTypePtr, 0) and
     (CurrentType <> ValPtype) then
    FlagErrorPos (318, row, col, fnum);
  if ContainsType (ResourceType, PTypePtr, 0) and
     (CurrentType <> ValPtype) then
    FlagErrorPos (319, row, col, fnum);
end else begin
  if CurrentType = ValPtype then begin
    if ContainsType (FileType, PTypePtr, 0) then
      FlagErrorPos (308, row, col, fnum);
    if ContainsType (TextType, PTypePtr, 0) then
      FlagErrorPos (308, row, col, fnum);
    if ContainsType (ProcessType, PTypePtr, 0) then
      FlagErrorPos (309, row, col, fnum);
    if ContainsType (ResourceType, PTypePtr, 0) then
      FlagErrorPos (310, row, col, fnum);
    if ContainsType (ChannelType, PTypePtr, 0) then
      FlagErrorPos (311, row, col, fnum);
  end;
end;
if not (LexSym.token in [semicolon, RightPar]) then FlagError (4);
if (LexSym.token = semicolon) then get;
for i := 1 to count do begin
  StoreParamData (PTypePtr, AddrOffset, CurrentType, IdNums[i]);
  if IsPar then ParamTabl ^ [ParamTablPtr].IsProcParm := true;
  DecParam (IdNums[i], ParamTablPtr, CurrentType);
  if IsPar or (CurrentType = ValPtype) then
    AddrOffset := AddrOffset + TypeTabl ^ [PTypePtr].TypeSize
  else AddrOffset := AddrOffset + 4;
  AddrOffset := (AddrOffset + 1) AND $fffffff;
end;
end;
expect(RightPar1,4);
for i := ParamTablPtr downto ParamTablPtr - TotalCount + 1 do begin
  if (ParamTabl^[i].IsVar=VarPtype) and (not IsPar) then
    ParamTabl^[i].AddrOffs := AddrOffset - ParamTabl^[i].AddrOffs - 4
  else
    ParamTabl^[i].AddrOffs := AddrOffset - ParamTabl^[i].AddrOffs -
      ((TypeTabl^[ParamTabl^[i].TypePointer].
        TypeSize + 1) and $fffffffe);
end;
TotParamSize := AddrOffset;
end;

begin
  ScanParmDec := ScanParmDecAct;
end.
unit sstat;
{used to scan all statements}

interface
uses
crt,
  labels, {used to generate internal labels and storage areas}
  lblock, {keeps track of current block level}
  getsym,lex, {used to obtain next token}
  types, {declares all types used}
  options, {used to flag all options - call to check options}
  tables, {stores all compiler created tables}
  gencode,codeemit, {used to generate psuedo code}
  CircRef, {used for procedural variables}
  errors; {used to flag all errors}

implementation

const
  NumericTypes : set of TypeDescType = [IntType,LongintType,ShortintType,
    ByteType,WordType,SubrangeType,
    RealType,DoubleType];

var
  ExprEnumTypePtr : integer;
  ExprPtrTypePtr : integer;
  FileTypePtr : integer;
  WithStackPtr : longint;

procedure ScanExpression(var ExprResult : ExprDataType);forward;

function numeric(s:ExprDataType):boolean;
begin
  numeric := s.ExprType in NumericTypes;
end;
procedure ScanEnumExpr(TypPtr: integer; var ExprResult: ExprDataType); forward;

procedure ScanArrayIndex(TypePtr: integer);
var ExprRsLt : ExprDataType;
begin
  if TypeTabl ^ [TypeTabl ^ [TypePtr].p1].TypeDesc = EnumType then
    ScanEnumExpr(TypeTabl ^ [TypePtr].p1,ExprRsLt)
  else ScanExpression(ExprRsLt);
  case ExprRsLt.ExprType of
    LongintType : if not (TypeTabl ^ [TypeTabl ^ [TypePtr].p1].TypeDesc in
       [IntType, LongintType, ShortintType, ByteType, WordType, SubrangeType]) then
      FlagError(139);
    CharType : if (TypeTabl ^ [TypeTabl ^ [TypePtr].p1].TypeDesc<>CharType) then
      FlagError(139);
    BoolType : if (TypeTabl ^ [TypeTabl ^ [TypePtr].p1].TypeDesc<>BoolType) then
      FlagError(139);
    EnumType : if (TypeTabl ^ [TypeTabl ^ [TypePtr].p1].TypeDesc<>EnumType) then
      FlagError(139);
    else FlagError(139);
  end;
  Emit(ChkRange, TypeTabl ^ [TypePtr].p1);
end;

procedure ScanVarAddressAct(var TypePtr: integer;
                          WritePossible, StopAtTemplate: boolean);
var IdData : IdRec;
  found : boolean;
  done : boolean;
  ExprRsLt : ExprDataType;
  i : integer;
  InBrkt : boolean;
  NumTemplateDimensions,
  ChkTempNum : word;
begin
  if LexSym.token = ptrl then begin
    get;
    ScanType(TypePtr);
    expect(LeftParl, 9);
    ScanExpression(ExprRsLt);
    if ExprRsLt.ExprType <> LongintType then
      FlagError(113);
    expect(RightParl, 4);
  end else begin

if LexSym.Token <> identifier1 then FlagError(2);
GetldDesc(LexSym.i1,IdData,found);
if not found then FlagError(104);
case IdData.IdTyp of
  ProcName : begin
    Emit(GetLabelAddress,ProcTabl^[IdData.valptr].CodeLabel);
    Get;
    TypePtr := -1;
    Exit;
  end;
  FuncName : begin
    Get;
    if LexSym.Token=LeftParl then begin
      { call the function }
      if TypeTabl^[FuncTabl^[IdData.valptr].TypePointer].TypeDesc
        <> PrType then FlagError(229);
      Emit(PrepFuncCall,IdData.valptr);
      with FuncTabl^[IdData.valptr] do
        ScanParams(FirstParam,ParameterCount,false);
      Expect(carrot1,230);
      TypePtr :=
        TypeTabl^[FuncTabl^[IdData.valptr].TypePointer].SubType;
      end else begin
      { assign a function result }
      if FuncTabl^[IdData.valptr].level >= BlockLevel then
        FlagError(177);
      Emit(PushBaseParamAddr,FuncTabl^[IdData.valptr].level+1);
      Emit(AddConstant,FuncTabl^[IdData.valptr].RsltOffset);
      TypePtr := FuncTabl^[IdData.valptr].TypePointer;
      end;
    end;
  end;
  VarName : begin
    case VarTabl^[IdData.valptr].IsAbsolute of
      NotAbs : begin
        Emit(PushBaseVarAddr,
          VarTabl^[IdData.valptr].level);
        Emit(AddConstant,
          VarTabl^[IdData.valptr].AddrOffset);
      end;
      AbsVar : begin
        if VarTabl^[IdData.valptr].AbsLevel=-1 then
          Emit(IntConstant,VarTabl^[IdData.valptr].AbsOffset)
        else begin
          Emit(PushBaseVarAddr,
            VarTabl^[IdData.valptr].AbsLevel);
          Emit(AddConstant,
            VarTabl^[IdData.valptr].AbsOffset);
        end;
    end;
  end;
AbsValParam : begin
  Emit(PushBaseParamAddr,
       VarTabl ^ [IdData.valptr].AbsLevel);
  Emit(AddConstant,
       VarTabl ^ [IdData.valptr].AbsOffset);
end;
AbsVarParam : begin
  Emit(PushBaseParamAddr,
       VarTabl ^ [IdData.valptr].AbsLevel);
  Emit(AddConstant,
       VarTabl ^ [IdData.valptr].AbsOffset);
  Emit(Fetch,LongintTypePtr);
end;
end;
ValParamName,
RetParamName : begin
  Emit(PushBaseParamAddr,IdData.level);
  Emit(AddConstant,ParamTabl ^ [IdData.valptr].AddrOffs);
  TypePtr := ParamTabl ^ [IdData.valptr].TypePointer;
end;
VarParamName : begin
  Emit(PushBaseParamAddr,IdData.level);
  Emit(AddConstant,ParamTabl ^ [IdData.valptr].AddrOffs);
  if not ParamTabl ^ [IdData.valptr].IsProcParm then
    if (ProcessLevelPtr = 0) or
        (ProcessLevel[ProcessLevelPtr] <>
         IdData.level) then
      Emit(Fetch,LongintTypePtr);
end;
end;
WithField : FlagError(398);
end;
end;
NumTemplateDimensions := 0;
ChkTempNum := 1;
done := false;
InBrkt := false;
repeat
  case TypeTabl ^ [TypePtr].TypeDesc of
    PtrType : begin
      if LexSym.token=carrot1 then begin
        get;
        Emit(Fetch,LongintTypePtr);
        TypePtr := TypeTabl ^ [TypePtr].SubType;
      end;
    end;
end;

if TypePtr = 0 then FlagError(134);
end else done := true;
end;

StringType : begin
  if LexSym.token=LeftBrkt1 then begin
    get;
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> LongintType then
      FlagError(113);
    Emit(ChkStrIndex,TypeTabl ^ [TypePtr].p1);
    Emit(Add,0);
    TypePtr := CharTypePtr;
    expect(RightBrkt1,12);
    end
  else done := true;
end;

RecordType : begin
  if LexSym.token=period1 then begin
    get;
    found := false;
    for i := TypeTabl ^ [TypePtr].p1 to
      TypeTabl ^ [TypePtr].p1 + TypeTabl ^ [TypePtr].p2-1
    do
      if FieldTabl ^ [i].FName=LexSym.il then begin
        found := true;
        Emit(AddConstant,FieldTabl ^ [i].AddrOffs);
        TypePtr := FieldTabl ^ [i].FTypePtr;
        end;
    if not found then FlagError(152);
    get;
    end
  else done := true;
end;

ArrayType : begin
  if LexSym.token=LeftBrkt1 then begin
    get;
    ScanArraylndex(TypePtr);
    if ChkTempNum <= NumTemplateDimensions then begin
      Emit(ChkTempIndex,ChkTempNum);
      inc(ChkTempNum);
      end;
    Emit(MultConstant,TypeTabl ^ [TypeTabl ^ [TypePtr].p2].
      TypeSize);
    Emit(Add,0);
    Emit(AddConstant,
         -(TypeTabl ^ [TypeTabl ^ [TypePtr].p1].p1 *
           TypeTabl ^ [TypeTabl ^ [TypePtr].p2].TypeSize));
    TypePtr := TypeTabl ^ [TypePtr].p2;
    if TypeTabl ^ [TypePtr].TypeDesc=ArrayType then begin
      if not(LexSym.token in [RightBrkt1,comma1]) then
        FlagError(12);
if LexSym.token=comma1 then
    LexSym.token := LeftBrkt1
else expect(RightBrkt1,12);
end
else expect(RightBrkt1,12);
end
else done := true;
end;

TemplateType : begin
    if StopAtTemplate then done := true
    else begin
        NumTemplateDimensions :=
            ArrayDepth(TypeTabl"^[TypePtr].SubType);
        ChkTempNum := 1;
        if WritePossible
            then Emit(PrepareTemplateWrite,TypePtr)
        else Emit(PrepareTemplateRead,TypePtr);
        if NumTemplateDimensions = 0 then
            Emit(FinishTemplateCheck,0);
        TypePtr := TypeTabl"^[TypePtr].SubType;
    end;
    else done := true;
end;
until done;
i := TypePtr;
while ChkTempNum <= NumTemplateDimensions do begin
    Emit(IntConstant,TypeTabl"^[TypeTabl"^[i].p1].p1);
    Emit(IntConstant,TypeTabl"^[TypeTabl"^[i].p1].p2);
    Emit(ChkTempIndexRow,ChkTempNum);
    inc(ChkTempNum);
i := TypeTabl"^[i].p2; {i points to element type}
end;
if NumTemplateDimensions <> 0 then Emit(FinishTemplateCheck,0);
end; {scan var address}

procedure ScanParamsAct(first,count:integer;IsProcess:boolean);
var i:integer;
VarType : integer;
ExprRslt : ExprDataType;
begin
    if count <> 0 then begin
        expect(LeftParl,9);
        for i := first to first+count-1 do begin
            if (ParamTabl"^[i].IsVar in [VarPtype,RetPtype]) OR
                ((TypeTabl"^[ParamTabl"^[i].TypePointer].TypeDesc in
                    [RecordType,ArrayType,ChannelType,SharedType,ResourceType])
                and IsProcess)
                then begin
                    ScanVarAddress(VarType,
                        ParamTabl"^[i].IsVar in [VarPtype,RetPtype],false);
                end;
        end;
    end;
if VarType <> ParamTabl^[i].TypePointer then
  FlagError(198);
  Emit(PushVar,0);
end else begin
  case TypeTabl^[ParamTabl^[i].TypePointer].TypeDesc of
    ArrayType,
    RecordType : begin
      ScanVarAddress(VarType,false,false);
      if VarType <> ParamTabl^[i].TypePointer then
        FlagError(198);
      Emit(PushStruct,TypeTabl^[ParamTabl^[i].
                     TypePointer].TypeSize);
    end;
    StringType : begin
      ScanExpression(ExprRslt);
      case ExprRslt.ExprType of
        CharType : begin
          Emit(PushCharAsStrParam,
               ParamTabl^[i].TypePointer);
        end;
        StringType : begin
          Emit(PushStrValParam,
               ParamTabl^[i].TypePointer);
        end;
      else FlagError(198);
    end;
  end;
  SetType : FlagError(398); {#### allow set parameters}
  EnumType : begin
    ScanEnumExpr(ParamTabl^[i].TypePointer,ExprRslt);
    Emit(PushVal,ParamTabl^[i].TypePointer);
  end;
  BoolType : begin
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> BoolType then
      FlagError(134);
    Emit(PushVal,ParamTabl^[i].TypePointer);
  end;
  PtrType : begin
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> PtrType then
      FlagError(134);
    Emit(PushVal,ParamTabl^[i].TypePointer);
  end;
  CharType : begin
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> CharType then
      FlagError(134);
    Emit(PushVal,ParamTabl^[i].TypePointer);
  end;
else begin
ScanExpression(ExprRslt);
if not numeric(ExprRslt) then
  FlagError(134);
  Emit(ChkRange,ParamTabl^ [i].TypePointer);
  Emit(PushVal,ParamTabl^ [i].TypePointer);
end;
end;
end;

if (i <> first+count-1) then
  expect(comma1,20);
end;

expect(RightPar1,4);
end;

procedure ScanEnumFuncCall;
var IdData : IdRec; found:boolean;
begin
  GetIdDesc(LexSym.il,IdData,found);
  Emit(PrepFuncCall,IdData.valptr);
  get;
  with FuncTabl^ [IdData.valptr] do
    ScanParams(FirstParam.ParameterCount, {true,} false);
  Emit(FuncCall,IdData.valptr);
end;

procedure ScanEnumExpr(TypPtr : integer;var ExprResult: ExprDataType);
var IsPred : boolean;
  IdData : IdRec;
  found : boolean;
  p1,p2 : integer;
  EnumFound : boolean;
  VarTypePtr : integer;
begin
  case LexSym.token of
    identifierl : begin
      GetIdDesc(LexSym.il,IdData,found);
      if (found) and (IdData.IdTyp = EnumConst) then begin
        LexSym.il := IdData.valptr;
        found := false;
      end;
      {look to see if the id is one of the enumerated constants}
      EnumFound := false;
      p1 := TypeTabl^ [TypPtr].p1;
      p2 := TypeTabl^ [TypPtr].p2;
      p2 := p1 + p2 - 1;
      while (not EnumFound) and (p1 <= p2) do
        if EnumTabl^ [p1]=LexSym.il then EnumFound := true
        else inc(p1);
      if EnumFound then begin
        ExprResult.ExprType := EnumType;
ExprResult.IsConst := true;
Emit(IntConstant,p1-TypeTabl ^ TypPtr].p1);
get;
end else begin
if not found then FlagError(104);
if not (IdData.IdTyp in [FuncName,
VarName,ValParamName,VarParamName,
RetParamName]) then
FlagError(212);
case IdData.IdTyp of
FuncName : begin
  if FuncTabl ^ [IdData.valptr].
    TypePointer <> TypPtr then
    FlagError(212);
  ScanEnumFuncCall;
  ExprResult.IsConst := false;
  end;
else begin
  ScanVarAddress(VarTypePtr,false,false);
  if VarTypePtr <> TypPtr then
    FlagError(212);
  Emit(Fetch,TypPtr);
  ExprResult.IsConst := false;
  end;
end;
end;
succ1,
pred1 : begin
  IsPred := LexSym.token=pred1;
  get;
  expect(LeftPar1,9);
  ScanEnumExpr(TypPtr,ExprResult);
  expect(RightPar1,4);
  if IsPred then Emit(AddConstant,1)
  else Emit(AddConstant,-1);
  ExprResult.IsConst := false;
  end;
else FlagError(212);
end;
ExprResult.ExprType := EnumType;
end;

procedure ScanExpression(var ExprResult : ExprDataType);

const
MaxStackLevel = 32;
IndexTypes : set of TypeDescType = [IntType,LongintType,ShortintType,
  ByteType,WordType,CharType,
  BoolType,EnumType,SubrangeType];
NumericTypes : set of TypeDescType = [IntType,LongintType,ShortintType,
type
  ExprStackType = array[0..MaxStackLevel] of ExprDataType;

var
  CmpOp : OpCode;
  StkPtr : integer;
  Stack : ExprStackType;
  EnumTypePointer : integer;

procedure IncStkPtr;
begin
  inc(StkPtr);
  if StkPtr > MaxStackLevel then FlagError(259);
  Stack[StkPtr].IsConst := false;
end;

procedure MarkConst;
begin
  Stack[StkPtr].IsConst := true;
end;

procedure DelConst;
begin
  Stack[StkPtr].IsConst := false;
end;

procedure ScanFuncCall;
var IdData : IdRec; found:boolean;
begin
  GetIdDescfLexSym.Id, IdData, found);
  if not found then FlagError(104);
  Emit(PrepFuncCall,IdData.valptr);
  get;
  with FuncTabl^[IdData.valptr] do
  begin
    ScanParams(FirstParam,ParameterCount,True,tfalse);
    Emit(FuncCall,IdData.valptr);
    IncStkPtr;
    Stack[StkPtr].ExprType := TypeTabl^[FuncTabl^[IdData.valptr].
      TypePointer].TypeDesc;
    if Stack[StkPtr].ExprType = PtrType then
      ExprPtrTypePtr := TypeTabl^[FuncTabl^[IdData.valptr].TypePointer].
        SubType;
    if Stack[StkPtr].ExprType in [ByteType,IntType,WordType,ShortintType]
      then Stack[StkPtr].ExprType := LongintType;
    if Stack[StkPtr].ExprType = DoubleType then
      Stack[StkPtr].ExprType := RealType;
    if Stack[StkPtr].ExprType = StringType then
      Emit(ExpandFuncStr,TypeTabl^[FuncTabl^[IdData.valptr].
procedure ScanFactor;
var found:boolean; IdData:IdRec;
    TypPtr :  integer;
    EnumTypePtr : integer;
    p1,p2 : integer;
    op : OpCode;
    ERslt: ExprDataType;

procedure ScanVarFactor;
begin
    ScanVarAddress(TypPtr,false,false);
    IncStkPtr;
    case TypeTabl' [TypPtr].TypeDesc of
        IntType,LongintType,ShortintType,SubrangeType,
        ByteType,WordType :  begin
            Stack[StkPtr].ExprType := LongintType;
            Emit(Fetch,TypPtr);
            end;
        RealType,DoubleType : begin
            Stack[StkPtr].ExprType := RealType;
            Emit(Fetch,TypPtr);
            end;
        PtrType : begin
            Stack[StkPtr].ExprType := PtrType;
            ExprPtrTypePtr := TypeTabl' [TypPtr].SubType;
            EmitCFetch,TypPtr);
            end;
        CharType : begin
            Stack[StkPtr] .ExprType := CharType;
            EmitCFetch,TypPtr);
            end;
        BoolType :  begin
            Stack[StkPtr].ExprType := BoolType;
            EmitCFetch,TypPtr);
            end;
        EnumType : begin
            Stack[StkPtr].ExprType := EnumType;
            ExprEnumTypePtr := TypPtr;
            EmitCFetch,TypPtr);
            end;
        StringType :  begin
            Stack[StkPtr] .ExprType := StringType;
            EmitCFetchStr,TypPtr);
            end;
        SetType :  FlagErrorC398); { # # #  allow sets}
        FileType : begin
            Stack[StkPtr].ExprType := FileType;
FileTypePtr := TypPtr;
end;

TextType : begin
    Stack[StkPtr].ExprType := TextType;
    FileTypePtr := TypPtr;
end;

ChannelType : begin
    Stack[StkPtr].ExprType := ChannelType;
    FileTypePtr := TypPtr;
end;

else FlagError(200);
end;
end;

begin {scan factor}
case LexSym.token of
  true1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := BoolType;
    MarkConst;
    Emit(BoolConstant, 1);
    get;
  end;
false1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := BoolType;
    MarkConst;
    Emit(BoolConstant, 0);
    get;
  end;
StringConst1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := StringType;
    MarkConst;
    Emit(StrConstant, LexSym.i1);
    get;
  end;
LongintConst1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := LongintType;
    MarkConst;
    Emit(IntConstant, LexSym.i1);
    get;
  end;
RealConst1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := RealType;
    MarkConst;
    Emit(RealConstant, LexSym.i1);
    get;
  end;
pi1 : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := RealType;
    MarkConst;
    Emit(RealConstant,1); {pi is constant # 1}
    get;
end;
ord1 : begin
    get;
    expect(LeftParl,9);
    IncStkPtr;
    ScanExpression(Stack[StkPtr]);
    if not (Stack[StkPtr].ExprType in [LongintType,
        CharType,BoolType,EnumType]) then FlagError(144);
    Stack[StkPtr].ExprType := LongintType;
    expect(RightParl,4);
end;
chr1 : begin
    get;
    expect(LeftParl,9);
    IncStkPtr;
    ScanExpression(Stack[StkPtr]);
    if not (Stack[StkPtr].ExprType in [LongintType,
        BoolType,EnumType]) then FlagError(144);
    Stack[StkPtr].ExprType := CharType;
    expect(RightParl,4);
end;
not1 : begin
    get;
    ScanFactor;
    if Stack[StkPtr].ExprType = BoolType then Emit(LogNot,0)
    else begin
        if Stack[StkPtr].ExprType <> LongintType then FlagError(65);
        Emit(BitwiseNot,0);
    end;
    end;
sin1,
cos1,
tan1,
asin1,
acos1,
atan1,
ln1,
sqrt1,
qr1,
round1,
exp1,
abs1 : begin
    case LexSym.token of
    sin1 : op := DoSin;
    cos1 : op := DoCos;
\begin{verbatim}
tan1 : op := DoTan;
asin1 : op := DoASin;
acos1 : op := DoACos;
atan1 : op := DoATan;
ln1 : op := DoLn;
exp1 : op := DoExp;
sqrt1 : op := DoSqrt;
sqr1 : op := DoSqr;
round1 : op := DoRound;
abs1 : op := DoAbs;
end;
get;
expect(LeftParl,9);
IncStkPtr;
ScanExpression(Stack[StkPtr]);
if NOT (Stack[StkPtr].ExprType in [RealType,LongintType])
  then FlagError(295);
expect(RightParl,4);
Emit(Op,0);
if op = DoRound then Stack[StkPtr].ExprType := LongintType
else Stack[StkPtr].ExprType := RealType;
end;
CharConst1 : begin
  IncStkPtr;
  Stack[StkPtr].ExprType := CharType;
  MarkConst;
  Emit(IntConstant,LexSym.i1);
  get;
  end;
LeftPar1 : begin
  get; \{ get rid of the '(' \}
  IncStkPtr;
  ScanExpression(Stack[StkPtr]);
  expect(RightParl,4);
  DelConst;
  end;
eof1 : begin
  get; \{ EOF \}
  Emit(SelectStdIO,0);
  if LexSym.token = LeftParl then begin
    Emit(ReleaseIO,0);
    get; \{ () \}
    ScanVarAddress(TypPtr,false,false);
    if not(TypeTabl ^ [TypPtr].TypeDesc in [TextType,FileType])
      then FlagError(216);
    Emit(SelectIO,0);
    expect(RightParl,4);
    end;
  Emit(ChkEOF.O);
  Emit(ReleaseIO,0);
  IncStkPtr;
\end{verbatim}
Stack[StkPtr].ExprType := BoolType;
DelConst;
end;
eocl : begin
get; {EOC}
expect(LeftParl,9);
ScanVarAddress(TypPtr,false,false);
if TypeTabl^[TypPtr].TypeDesc <> ChannelType then
FlagError(216);
expect(RightParl,4);
Emit(ChkEOC,0);
IncStkPtr;
Stack[StkPtr].ExprType := BoolType;
DelConst;
end;
addrl : begin
get; {ADD}
expect(LeftParl,9);
ScanVarAddress(TypPtr,false,false);
expect(RightParl,4);
IncStkPtr;
Stack[StkPtr].ExprType := LongintType;
DelConst;
Emit(Address2Int,0);
end;
sizeofl : begin
get; {sizeof}
expect(LeftParl,9);
ScanType(TypPtr);
expect(RightParl,4);
Emit(IntConstant,TypeTabl^[TypPtr].TypeSize);
IncStkPtr;
Stack[StkPtr].ExprType := LongintType;
end;
identifierl : begin
GetIdDesc(LexSym.i1,IdData,found);
if found then
case IdData.IdTyp of
  LongintConst : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := LongintType;
    MarkConst;
    Emit(IntConstant,IdData.valptr);
    get;
    end;
  CharConst : begin
    IncStkPtr;
    Stack[StkPtr].ExprType := CharType;
    MarkConst;
    Emit(IntConstant,IdData.valptr);
    get;
end;
end;

StrConst : begin
  IncStkPtr;
  Stack[StkPtr].ExprType := StringType;
  MarkConst;
  Emit(StrConstant,IdData.valptr);
end;

RealConst : begin
  IncStkPtr;
  Stack[StkPtr].ExprType := RealType;
  MarkConst;
  Emit(RealConstant,IdData.valptr);
end;

BoolConst : begin
  IncStkPtr;
  Stack[StkPtr].ExprType := BoolType;
  MarkConst;
  Emit(IntConstant,IdData.valptr);
end;

EnumConst : begin
  IncStkPtr;
  Stack[StkPtr].ExprType := EnumType;
  MarkConst;
  Emit(IntConstant,IdData.valptr);
end;

TypeName : FlagError(398);
FuncName,
FwdFuncName : begin
  ScanFuncCall;
  if (Stack[StkPtr].ExprType = PtrType) AND
    (LexSym.Token=carrot1) then begin
    if not (TypeTabl^[ExprPtrTypePtr].TypeDesc in
      [IntType,LongintType,ShortintType,ByteType,WordType,
      RealType,DoubleType,SingleType,PtrType,FwdPtrType,
      CharType,BoolType,StringType,SubrangeType,EnumType])
      then FlagError(58);
    Emit(Fetch,ExprPtrTypePtr);
  end;
end;

VarName,
ValParamName,
RetParamName,
VarParamName : ScanVarFactor;
else FlagError(58);
end
else begin
  found := false;
EnumTypePtr := TypeTablPtr;
while (not found) and (EnumTypePtr>0) do begin
  if TypeTabl^[EnumTypePtr].TypeDesc=EnumType then begin
    p1 := TypeTabl^[EnumTypePtr].p1;
p2 := TypeTabl^[EnumTypePtr].p2;
p2 := p1 + p2 - 1;
    while (not found) and (p1 <= p2) do
      if EnumTabl^[p1] = LexSym.i1 then found := true
        else inc(p1);
  end else dec(EnumTypePtr);
if not found then FlagError(104);
ExprEnumTypePtr := EnumTypePtr;
p1 := p1 - TypeTabl^[EnumTypePtr].p1;
Emit(IntConstant,p1);
IncStkPtr;
Stack[StkPtr].ExprType := EnumType;
MarkConst;
get;
end;
end;

ptr1 : ScanVarFactor;
and1 : begin
  Get;
  ScanVarAddress(TypPtr, false, false);
  IncStkPtr;
  Stack[StkPtr].ExprType := PtrType;
  ExprPtrTypePtr := TypPtr;
end;
nill : begin
  Get;
  Emit(IntConstant,0);
  IncStkPtr;
  Stack[StkPtr].ExprType := PtrType;
  ExprPtrTypePtr := 0;
end;
else FlagError(58);
end;

procedure ScanTerm;
var temp : LexOutput; ResultType : integer;
  AndFailLabel : longint;
begin
  ScanFactor;
  if (LexSym.token=and1) and (Stack[StkPtr].ExprType=BoolType) then begin
    AndFailLabel := NewLabel;
    Emit(JmpFalse, AndFailLabel);
    while LexSym.token = and1 do begin
      get; {AND}
ScanFactor;
if (Stack[StkPtr].ExprType <> BoolType) then FlagError(401);
Emit(JmpFalse,AndFailLabel);
{PropagateFuncCalls;}
dec(StkPtr);
DelConst;
end;
Emit(EndOfAnd,AndFailLabel);
ForgetLabel(AndFailLabel);
end
else while (LexSym.token in [timesl.dividel.divl.modl.andl]) do begin
  temp := LexSym;  {save the term op here}
  get;  {move on}
  ScanFactor;
  case temp.token of
    andl : begin
      if ((Stack[StkPtr].ExprType=LongintType) and
          (Stack[StkPtr-1].ExprType=LongintType)) OR
          ((Stack[StkPtr].ExprType=BoolType) and
           (Stack[StkPtr-1].ExprType=BoolType)) then
        Emit(LogAnd.O)
      else FlagError(134);
    {PropagateFuncCalls;}
dec(StkPtr);
    DelConst;
end;
timesl : begin
  if (numeric(Stack[StkPtr]) and numeric(Stack[StkPtr-1])) then begin
    Emit(Mult,0);
    if (Stack[StkPtr].ExprType=RealType) OR
        (Stack[StkPtr-1].ExprType=RealType) then
      Stack[StkPtr-1].ExprType := RealType
    else Stack[StkPtr-1].ExprType := LongintType;
    {PropagateFuncCalls;}
dec(StkPtr);
    DelConst;
end else FlagError(197);
end;
divl : begin
  if (Stack[StkPtr].ExprType=LongintType) and
      (Stack[StkPtr-1].ExprType=LongintType) then begin
    Emit(IntDiv,0);
    {PropagateFuncCalls;}
dec(StkPtr);
    DelConst;
end else FlagError(197);
end;
dividel : begin
  if (numeric(Stack[StkPtr]) and numeric(Stack[StkPtr-1])) then begin
    Emit(Divide,0);
    Stack[StkPtr-1].ExprType := RealType;
end;
procedure ScanSexpr;
var DoNegate:boolean; temp:LexOutput; ResultType : integer;
SuccessLabel : longint;
begin
  DoNegate := false;
  if (LexSym.token=minusl) then begin
    DoNegate := true; {indicate the negation to take place}
    get; {get rid of the minus sign}
  end else if (LexSym.token=plusl) then get; {this is a NOP}
  ScanTerm;
  if DoNegate then begin
    if numeric(Stack[StkPtr]) then Emit(NegateVal,0)
    else FlagError(196);
  end;
  if (LexSym.token = orl) and (Stack[StkPtr].ExprType=BoolType) then begin
    SuccessLabel := NewLabel;
    Emit(JmpTrue,SuccessLabel);
    while LexSym.token = orl do begin
      get; {OR}
      ScanTerm;
      if (Stack[StkPtr].ExprType <> BoolType) then FlagError(401);
      Emit(JmpTrue,SuccessLabel);
      {PropagateFuncCalls;}
      dec(StkPtr);
      DelConst;
    end;
    Emit(EndOfOr,SuccessLabel);
    ForgetLabel(SuccessLabel);
  end else while (LexSym.token in [plusl, minusl, orl, xorl]) do begin
    temp := LexSym; {save the sexpr op}
    get; {move on}
    ScanTerm;
    case temp.token of
orl : begin
    if ((Stack[StkPtr].ExprType=LongintType) and
        (Stack[StkPtr-1].ExprType=LongintType)) OR
    ((Stack[StkPtr].ExprType=BoolType) and
     (Stack[StkPtr-1].ExprType=BoolType)) then
        Emit(LogOr,0)
    else FlagError(134);
    {PropagateFuncCalls;}
    dec(StkPtr);
    DelConst;
    end;

xorl : begin
    if ((Stack[StkPtr].ExprType=LongintType) and
        (Stack[StkPtr-1].ExprType=LongintType)) OR
    ((Stack[StkPtr].ExprType=BoolType) and
     (Stack[StkPtr-1].ExprType=BoolType)) then
        Emit(LogXor,0)
    else FlagError(134);
    {PropagateFuncCalls;}
    dec(StkPtr);
    DelConst;
    end;

minus1 : begin
    if (numeric(Stack[StkPtr])) and (numeric(Stack[StkPtr-1]))
    then begin
        Emit(Subtract,0);
        if (Stack[StkPtr].ExprType=RealType) or
            (Stack[StkPtr-1].ExprType=RealType) then
            Stack[StkPtr-1].ExprType := RealType
        else
            Stack[StkPtr-1].ExprType := LongintType;
        {PropagateFuncCalls;}
        dec(StkPtr);
        DelConst;
    end else begin
        Error(134);
    end;

plus1 : begin
    if (numeric(Stack[StkPtr])) and (numeric(Stack[StkPtr-1]))
    then begin
        Emit(ADD,0);
        if (Stack[StkPtr].ExprType=RealType) or
            (Stack[StkPtr-1].ExprType=RealType) then
            Stack[StkPtr-1].ExprType := RealType
        else
            Stack[StkPtr-1].ExprType := LongintType;
        {PropagateFuncCalls;}
        dec(StkPtr);
        DelConst;
    end else begin
        Error(134);
    end;
then
begin
if (Stack[StkPtr-1].ExprType = CharType) and
   (Stack[StkPtr].ExprType = CharType) then
   Emit(AddCharChar,0);
if (Stack[StkPtr-1].ExprType = CharType) and
   (Stack[StkPtr].ExprType = StringType) then
   Emit(AddCharStr,0);
if (Stack[StkPtr-1].ExprType = StringType) and
   (Stack[StkPtr].ExprType = CharType) then
   Emit(AddStrChar,0);
if (Stack[StkPtr-1].ExprType = StringType) and
   (Stack[StkPtr].ExprType = StringType) then
   Emit(AddStrings,0);
{PropagateFuncCalls;}
dec (StkPtr);
Stack[StkPtr].ExprType := StringType;
DelConst;
end else { # # # # check for sets here}
FlagError(134);
end
end
end
end

var
LPtrTypePtr : integer;

begin {ScanExpression}
StkPtr := -1;
ScanSexpr;
if (LexSym.token in [equal,NotEqual,LessThen1,
   GreaterThen1,LessEq1,GreaterEq1]) then begin
if Stack[StkPtr].ExprType=EnumType then begin
   case LexSym.token of
      equal : CmpOp := CmpEQ;
      NotEqual : CmpOp := CmpNE;
      else FlagError(213);
   end;
get;
EnumTypePointer := ExprEnumTypePtr;
IncStkPtr;
ScanEnumExpr(EnumTypePointer,Stack[StkPtr]);
Emit(CmpOp,EnumTypePointer);
{PropagateFuncCalls;}
dec(StkPtr);
DelConst;
Stack[StkPtr].ExprType := BoolType;
end else begin
   case LexSym.token of
equal1 : CmpOp := CmpEQ;
NotEqual1 : CmpOp := CmpNE;
GreaterThen1 : CmpOp := CmpGT;
LessThen1 : CmpOp := CmpLT;
LessEq1 : CmpOp := CmpLE;
GreaterEq1 : CmpOp := CmpGE;
end;

get; {get the next token}
LPtrTypePtr := ExprPtrTypePtr;
ScanSexpr;
if (numeric(Stack[StkPtr]) and numeric(Stack[StkPtr-l])) OR
((Stack[StkPtr].ExprType = Stack[StkPtr-1].ExprType) AND
(Stack[StkPtr].ExprType in [CharType,BoolType.PtrType])
then
begin
if (Stack[StkPtr].ExprType = PtrType)
AND (LPtrTypePtr <> ExprPtrTypePtr)
AND (LPtrTypePtr <> 0) AND (ExprPtrTypePtr <> 0)
then FlagError(195);
Emit(CmpOp,0);
dec(StkPtr);
DelConst;
Stack[StkPtr].ExprType := BoolType;
end else begin
if Stack[StkPtr-1].ExprType in [CharType,StringType] then begin
if not (Stack[StkPtr].ExprType in [CharType,StringType]) then
FlagError(225);
if Stack[StkPtr].ExprType = Stack[StkPtr-1].ExprType then
case CmpOp of
CmpLT : Emit(CmpStrs,1);
CmpLE : Emit(CmpStrs,2);
CmpEQ : Emit(CmpStrs,3);
CmpNE : Emit(CmpStrs,4);
CmpGE : Emit(CmpStrs,5);
CmpGT : Emit(CmpStrs,6);
end
else begin
if Stack[StkPtr].ExprType = CharType then
case CmpOp of
CmpLT : Emit(CmpStrChar,1);
CmpLE : Emit(CmpStrChar,2);
CmpEQ : Emit(CmpStrChar,3);
CmpNE : Emit(CmpStrChar,4);
CmpGE : Emit(CmpStrChar,5);
CmpGT : Emit(CmpStrChar,6);
end
else
case CmpOp of
CmpLT : Emit(CmpCharStr,5);
CmpLE : Emit(CmpCharStr,6);
CmpEQ : Emit(CmpCharStr,3);
CmpNE : Emit(CmpCharStr,4);
CmpGE : Emit(CmpCharStr,1);
CmpGT : Emit(CmpCharStr,2);
end;

{PropagateFuncCalls;}
dec(StkPtr);
DelConst;
Stack[StkPtr].ExprType := BoolType;
end
else FlagError(195);
end;
end;
end;
ExprResult := Stack[0];
dec(StkPtr);
end;

procedure ScanProcCall;
var IdData:IdRec; found:boolean;
begin
Emit(lnitialize,0);
GetldDesc(LexSym.il,IdData,found);
if not found then FlagError(104);
get;
with ProcTabl'[IdData.valptr] do
  ScanParams(FirstParam,ParameterCount,false);
Emit(ProcCall,IdData.valptr);
end;

procedure ScanAssign;
var LTypePointer, RtypePointer:integer;
ExprRslt : ExprDataType;
i : longint;
begin
Emit(lnitialize,0);
ScanVarAddress(LTypePointer,true,false);
if not (TypeTabl'[LTypePointer].TypeDesc in [ResourceType,ProcessType])
  then expect(assignl,51);
case TypeTabl'[LTypePointer].TypeDesc of
  SharedType,
  TemplateType,
  ChannelType : FlagError(406);
  StringType : begin
    ScanExpression(ExprRslt);
case ExprRslt.ExprType of
    CharType : Emit(StoreCharlnStr,0);
    StringType : Emit(StoreStr,LTypePointer);
    else FlagError(129);
    end;
  end;
  EnumType : begin
ScanEnumExpr(LTypePointer,ExprRslt);
Emit(ChkRange,LTypePointer);
Emit(Store,LTypePointer);
end;

SetType : FlagError(398); {### allow sets}

BoolType : begin
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> BoolType then FlagError(129);
    Emit(Store,LTypePointer);
end;

CharType : begin
    ScanExpression(ExprRslt);
    if ExprRslt.ExprType <> CharType then FlagError(129);
    Emit(Store,LTypePointer);
end;

PtrType : begin
    ScanExpression(ExprRslt);
    if (ExprRslt.ExprType <> PtrType)
        OR ((ExprPtrTypePtr <> TypeTabl[LTypePointer].SubType)
            AND (ExprPtrTypePtr <> 0))
    then FlagError(129);
    Emit(Store,LTypePointer);
end;

TextType,
FileType : FlagError(146);

ArrayType,
RecordType : begin
    ScanVarAddress(RtypePointer,false,false);
    if RtypePointer <> LTypePointer then FlagError(129);
    Emit(CopyStruct,TypeTabl[RtypePointer].TypeSize);
end;

ResourceType : begin
    { The resource address is on the stack. }
    { Keep it there until after the params are scanned. }
    expect(period1,23);
    if LexSym.token <> identifier1 then FlagError(2);
    i := TypeTabl[LTypePointer].p1;
    while (LexSym.il <> FieldTabl[i].FName) and
        (i < (TypeTabl[LTypePointer].p1 + TypeTabl[LTypePointer].p2)) do
        inc(i);
    if (i >= TypeTabl[LTypePointer].p1 + TypeTabl[LTypePointer].p2)
        then FlagError(294);
    get;
    ScanParams(FieldTabl[i].FTypePtr,
        FieldTabl[i].FTypePcnt.true);
    { Now, fetch the contents of the resource variable. }
    Emit(Fetch,LongintTypePtr);
    { And push the resource ID onto the stack. }
    Emit(PushVal,LongintTypePtr);
    Emit(GetLabelAddress,FieldTabl[i].AddrOffs);
    Emit(CallRsrcProc,0);
end
ProcessType : FlagError(293);
else begin {all the numeric types}
  ScanExpression(ExprRslt);
  if not numeric(ExprRslt) then FlagError(129);
  Emit(ChkRange,LTypePointer);
  Emit(Store,LTypePointer);
end;
end;
end;

procedure ScanSmpStat;
var IdDescript : IdRec; IdFound:boolean;
begin
  if LexSym.token = ptrl then ScanAssign
  else begin
    GetIdDesc(LexSym.il,IdDescript,IdFound);
    if not IdFound then FlagError(104);
    case IdDescript.IdTyp of
      ProcName,FwdProcName :  ScanProcCall;
      FuncName,
      VarName,ValParamName,
      VarParamName, RetParamName :  ScanAssign;
      DecRsrcProc,UsedRsrcProc : FlagError(292);
      FwdFuncName :  FlagError(177);
      else FlagError(60);
    end;
  end;
end;

procedure ScanIfStat;
var ElseLabel,EndifLabel:longint;
ExprRslt : ExprDataType;
begin
  get; {IF}
  ElseLabel := NewLabel;
  ScanExpression(ExprRslt);
  if (ExprRslt.ExprType <> BoolType) then FlagError(135);
  Emit(Jmpfalse,ElseLabel);
  expect(thenl,52);
  ScanStat;
  if LexSym.token=else1 then begin
    get; {ELSE}
    EndifLabel := NewLabel;
    Emit(Jmpalways,EndifLabel);
    Emit(declabel,ElseLabel);
    ScanStat;
    Emit(declabel,EndifLabel);
    ForgetLabel(EndifLabel);
  end else Emit(declabel,ElseLabel);
  ForgetLabel(ElseLabel);
procedure ScanForStatAct(IsParallel : boolean);
var ForLabel, FinishLabel : longint;
  ExprRslt : ExprDataType;
  TypePtr : integer;
  IsDownto : boolean;
begin
  get; {for}
  ScanVarAddress(TypePtr, true, false);
  if not((TypeTabl^[TypePtr].TypeDesc in [LongintType, IntType, WordType, ByteType, ShortintType, EnumType, SubrangeType, BoolType, CharType])
  then FlagError(210);
  if TypeTabl^[TypePtr].TypeDesc=EnumType then
    ScanEnumExpr(TypePtr, ExprRslt)
  else ScanExpression(ExprRslt);
  Emit(ChkRange,TypePtr);
  case TypeTabl^[TypePtr].TypeDesc of
  LongintType, IntType, WordType,
    ByteType, ShortintType, SubrangeType : if ExprRslt.ExprType <> LongintType then
      FlagError(211);
    CharType : if ExprRslt.ExprType <> CharType then FlagError(211);
    BoolType : if ExprRslt.ExprType <> BoolType then FlagError(211);
    EnumType : {already checked by ScanEnumExpr} begin end;
  end;
  if not (LexSym.token in [to1, downto1]) then FlagError(55);
  IsDownto := LexSym.token=downto1;
  get;
  if TypeTabl^[TypePtr].TypeDesc=EnumType then
    ScanEnumExpr(TypePtr, ExprRslt)
  else ScanExpression(ExprRslt);
  Emit(ChkRange, TypePtr);
  case TypeTabl^[TypePtr].TypeDesc of
  LongintType, IntType, WordType,
    ByteType, ShortintType, SubrangeType : if ExprRslt.ExprType <> LongintType then
      FlagError(211);
    CharType : if ExprRslt.ExprType <> CharType then FlagError(211);
    BoolType : if ExprRslt.ExprType <> BoolType then FlagError(211);
    EnumType : {already checked by ScanEnumExpr} begin end;
  end;
  if IsParallel then Emit(EnablePar, 0) else Emit(DisablePar, 0);
  Emit(ForPrelim, TypePtr); {store the initial value, push address and count}
  ForLabel := NewLabel;
  FinishLabel := NewLabel;
  if IsDownto then Emit(DowntoCmp, FinishLabel)
  else Emit(ToCmp, FinishLabel);
  expect(dol, 54);
  Emit(DecLabel, ForLabel);
if IsParallel then begin
  if LexSym.token = for1 then ScanForStat(true)
  else ScanProcessCall;
end else ScanStat;
if IsParallel then Emit(EnablePar,0) else Emit(DisablePar,0);
if IsDownto then Emit(ForDec,TypePtr)
else Emit(ForInc,TypePtr);
Emit(JmpTrue,ForLabel);
Emit(DecLabel,FinishLabel);
if IsParallel then Emit(EnablePar,0) else Emit(DisablePar,0);
Emit(ForEnd,0);
ForgetLabel(ForLabel);
ForgetLabel(FinishLabel);
end;

procedure ScanWhileStat;
var WhileLabel,FinishLabel : longint;
  ExprRslt : ExprDataType;
begin
  WhileLabel := NewLabel;
  FinishLabel := NewLabel;
  Emit(declabel,WhileLabel);
  get; {WHILE}
  ScanExpression(ExprRslt);
  if (ExprRslt.ExprType <> BoolType) then FlagError(135);
  Emit(jmpfalse,FinishLabel);
  expect(dol,54);
  ScanStat;
  Emit(jmpalways,WhileLabel);
  Emit(declabel,FinishLabel);
  ForgetLabel(WhileLabel);
  ForgetLabel(FinishLabel);
end;

procedure ScanCaseStat;
var EndLabel,NextLabel,ThisLabel : longint;
  ExprRslt,CexprRslt : ExprDataType;
  EnumTypPtr : integer;
  IsEnum : boolean;
begin
  EndLabel := NewLabel;
  get; {clear out the case}
  ScanExpression(ExprRslt);
  IsEnum := false;
  if ExprRslt.ExprType = EnumType then begin
    EnumTypPtr := ExprEnumTypePtr;
    Emit(ChkRange,EnumTypPtr);
    IsEnum := true;
  end;
  if not (ExprRslt.ExprType in [LongintType,CharType,BoolType,EnumType])
  then FlagError(208);
Emit(EvalCase,0);
expect(of1,8);
while not (LexSym.token in [else1,end1]) do begin
  NextLabel := NewLabel;
  ThisLabel := NewLabel;
  if IsEnum then ScanEnumExpr(EnumTypPtr,CexprRslt)
  else ScanExpression(CexprRslt);
  if not CexprRslt.IsConst then FlagError(209);
  if CexprRslt.ExprType <> ExprRslt.ExprType then FlagError(129);
  if LexSym.token = colon1 then Emit(CaseCmp,-NextLabel)
  else Emit(CaseCmp,ThisLabel);
  while LexSym.token=comma1 do begin
    get;
    if IsEnum then ScanEnumExpr(EnumTypPtr,CexprRslt)
    else ScanExpression(CexprRslt);
    if not CexprRslt.IsConst then FlagError(209);
    if CexprRslt.ExprType <> ExprRslt.ExprType then FlagError(129);
    if LexSym.token = colon1 then Emit(CaseCmp,-NextLabel)
    else Emit(CaseCmp,ThisLabel);
  end;
  Emit(DecLabel,ThisLabel);
  ForgetLabel(ThisLabel);
  expect(colon1,5);
  ScanStat;
  if LexSym.token <> end1 then expect(semicolon1,14);
  Emit(JmpAlways,EndLabel);
  Emit(DecLabel,NextLabel);
  ForgetLabel(NextLabel);
end;
if LexSym.token=else1 then begin
  get;
  ScanStat;
  if LexSym.token=semicolon1 then get;
end;
  Emit(DecLabel,EndLabel);
  ForgetLabel(EndLabel);
  expect(end1,13);
end;

procedure ScanRepeatStat;
var RepeatLabel : longint;
  ExprRslt : ExprDataType;
begin
  get; {REPEAT}
  RepeatLabel := NewLabel;
  Emit(declabel,RepeatLabel);
  ScanStat;
  while LexSym.token=semicolon1 do begin
    get; {then semicolon}
    ScanStat;
  end;
end;
expect(until1,53);
ScanExpression(ExprRslt);
if (ExprRslt.ExprType <> BoolType) then FlagError(135);
Emit(jmpfalse,RepeatLabel);
ForgetLabel(RepeatLabel);
end;

procedure ScanCmpdStatAct;
begin
get; {clear out the BEGIN}
ScanStat; {scan the first statement}
while LexSym.token = semicolon1 do begin
  get; {clear out the ;}
  ScanStat; {scan all subsequent statements}
end;
if LexSym.token <> endl then FlagError(14);
expect(endl,13);
end;

procedure ScanloStat;
var ExprRslt, FormatExprRslt : ExprDataType;
  FilePtr, DataPtr : integer;
begin
  case LexSym.token of
    fassign1 : begin
      get; {assign}
      expect(LeftPar1,9);
      ScanExpression(ExprRslt);
      if not (ExprRslt.ExprType in [FileType.TextType]) then
        FlagError(216);
      FilePtr := FileTypePtr;
      expect(commal,20);
      ScanExpression(ExprRslt);
      if ExprRslt.ExprType=CharType then begin
        Emit(CharToStr,0);
        ExprRslt.ExprType := StringType;
      end;
      if ExprRslt.ExprType <> StringType then
        FlagError(217);
      Emit(AssignFile,FilePtr);
      expect(RightParl,4);
    end;
    reset1 : begin
      get; {reset}
      expect(LeftPar1,9);
      ScanExpression(ExprRslt);
      if not (ExprRslt.ExprType in [FileType.TextType, ChannelType]) then FlagError(216);
      FilePtr := FileTypePtr;
      Emit(ResetFile,FilePtr);
      expect(RightParl,4);
    end;
  end;
end;
end;

rewrite1 : begin
  get; {rewrite}
  expect(LexPar1,9);
  ScanExpression(ExprRslt);
  if not (ExprRslt.ExprType in [FileType,TextType, ChannelType]) then FlagError(216);
  FilePtr := FileTypePtr;
  Emit(RewriteFile,FilePtr);
  expect(RightPar1,4);
end:

close1 : begin
  get; {close}
  expect(LexPar1,9);
  ScanExpression(ExprRslt);
  if not (ExprRslt.ExprType in [FileType,TextType, ChannelType]) then FlagError(216);
  FilePtr := FileTypePtr;
  if ExprRslt.ExprType=ChannelType then begin
    expect(comma1,20);
    ScanVarAddress(DataPtr,false,false);
    if DataPtr <> TypeTabl^[FilePtr].SubType then
      FlagError(220);
  end;
  Emit(CloseFile,FilePtr);
  expect(RightPar1,4);
end:

seek1 : begin
  get; {seek}
  expect(LexPar1,9);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType = TextType then
    FlagError(218);
  if ExprRslt.ExprType <> FileType then
    FlagError(216);
  FilePtr := FileTypePtr;
  expect(comma1,20);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> LongintType then
    FlagError(219);
  Emit(SeekFile,FilePtr);
  expect(RightPar1,4);
end:

read1 : begin
  get; {read}
  Emit(SelectStdIO,0);
  expect(LexPar1,9);
  {check for a file here}
  ScanVarAddress(FilePtr,true,false);
  if TypeTabl^[FilePtr].TypeDesc in [FileType,ChannelType]
    then begin
Emit(ReleaseIO,0);
{if TypeTabl^[FilePtr].TypeDesc = FileType then
  FilePtr := FileTypePtr;
} expect(comma1,20);
ScanVarAddress(DataPtr,true,false);
if DataPtr <> TypeTabl^[FilePtr].SubType then
  FlagError(220);
Emit(ReadFile,FilePtr);
expect(RightPar1,4);
end else begin
if TypeTabl^[FilePtr].TypeDesc = TextType then begin
  Emit(ReleaseIO,0);
  Emit(SelectIO,0);
  expect(comma1,20);
  ScanVarAddress(DataPtr,true,false);
case TypeTabl^[DataPtr].TypeDesc of
  LongintType,IntType,
  WordType,ShortintType,
  ByteType,SubrangeType,
  RealType,DoubleType,
  CharType,StringType : Emit(ReadVar,DataPtr);
else FlagError(224);
end;
while LexSym.token=comma1 do begin
  get; {},
  ScanVarAddress(DataPtr,true,false);
case TypeTabl^[DataPtr].TypeDesc of
  LongintType,IntType,
  WordType,ShortintType,
  ByteType,SubrangeType,
  RealType,DoubleType,
  CharType,StringType : Emit(ReadVar,DataPtr);
else FlagError(224);
end;
end;
expect(RightPar1,4);
Emit(ReleaseIO,0);
end else begin
DataPtr := FilePtr; {no text file specified - STD IO}
case TypeTabl^[DataPtr].TypeDesc of
  LongintType,IntType,
  WordType,ShortintType,
  ByteType,SubrangeType,
  RealType,DoubleType,
  CharType,StringType : Emit(ReadVar,DataPtr);
else FlagError(224);
end;
while LexSym.token=comma1 do begin
  get; {},
  ScanVarAddress(DataPtr,true,false);
case TypeTabl^[DataPtr].TypeDesc of
LongintType, IntType,
WordType, ShortintType,
ByteType, SubrangeType,
RealType, DoubleType,
CharType, StringType : Emit(ReadVar, DataPtr);
else FlagError(224);
end;
end;
expect(RightPar, 4);
Emit(ReleaseIO, 0);
end;
end;
end;

readln1 : begin
get; {readln}
Emit(SelectStdIO, 0);
if LexSym.token = LeftPar then begin
get; { ( }
{check for a text file here}
ScanVarAddress(FilePtr, true, false);
if TypeTable$[FilePtr].TypeDesc = FileType then
FlagError(221);
if TypeTable$[FilePtr].TypeDesc = TextType then begin
Emit(ReleaseIO, 0);
Emit(SelectIO, 0);
while LexSym.token = comma do begin
get; {,}
ScanVarAddress(DataPtr, true, false);
case TypeTable$[DataPtr].TypeDesc of
LongintType, IntType,
WordType, ShortintType,
ByteType, SubrangeType,
RealType, DoubleType,
CharType, StringType : Emit(ReadVar, DataPtr);
else FlagError(224);
end;
end
end else begin
DataPtr := FilePtr; {it's an input variable, not a file}
case TypeTable$[DataPtr].TypeDesc of
LongintType, IntType,
WordType, ShortintType,
ByteType, SubrangeType,
RealType, DoubleType,
CharType, StringType : Emit(ReadVar, DataPtr);
else FlagError(224);
end;
while LexSym.token = comma do begin
get; {,}
ScanVarAddress(DataPtr, true, false);
case TypeTable$[DataPtr].TypeDesc of
LongintType, IntType,
WordType, ShortintType,
ByteType, SubrangeType,
RealType, DoubleType,
CharType, StringType : Emit(ReadVar, DataPtr);
else FlagError(224);
end;
end;
end;
expect(RightParl,4);
end;
Emit(ClrInputLine,0);
Emit(ReleaseIO,0);
end;
write1 : begin
get; {write}
Emit(SelectStdIO,0);
expect(LeftParl,9);
ScanExpression(ExprRslt);
if ExprRslt.ExprType in [FileType, ChannelType] then begin
  Emit(ReleaseIO,0);
  FilePtr := FileTypePtr;
  expect(comma1,20);
  ScanVarAddress(DataPtr, false, false);
  if DataPtr <> TypeTabl''[FilePtr].SubType then
    FlagError(220);
  Emit(WriteFile, FilePtr);
end else begin
  if ExprRslt.ExprType=TextType then begin
    Emit(ReleaseIO,0);
    Emit(SelectIO,0);
    if LexSym.token=comma1 then FlagError(20);
  end else begin
    if not(ExprRslt.ExprType in [LongintType, RealType,
       CharType, StringType]) then FlagError(222);
    if LexSym.token=colon1 then begin
      get;{:}
      ScanExpression(FormatExprRslt);
      if FormatExprRslt.ExprType <> LongintType then
        FlagError(223);
    end else Emit(IntConstant,-1);
    if ExprRslt.ExprType=RealType then begin
      if LexSym.token=colon1 then begin
        get;{:}
        ScanExpression(FormatExprRslt);
        if FormatExprRslt.ExprType <> LongintType then
          FlagError(223);
      end else Emit(IntConstant,-1);
    end;
case ExprRslt.ExprType of
  LongintType : Emit(PrintInt,0);
RealType : Emit(PrintReal,0);
CharType : Emit(PrintChar,0);
StringType : Emit(PrintString,0);
end;
end;
while LexSym.token=comma1 do begin
get; 
ScanExpression(ExprRslt);
if not(ExprRslt.ExprType in [LongintType,RealType,
CharType,StringType]) then FlagError(222);
if LexSym.token=colon1 then begin
get;
ScanExpression(FormatExprRslt);
if FormatExprRslt.ExprType <> LongintType then
FlagError(223);
end else Emit(IntConstant,-1);
end;
end;
case ExprRslt.ExprType of
LongintType : Emit(PrintInt,O);
RealType : Emit(PrintReal,O);
CharType : Emit(PrintChar,O);
StringType : Emit(PrintString,O);
end;
end;
Emit(Flush1O,O);
Emit(ReleaseIO,O);
end;
expect(RightParl,4);
end;
writeln1 : begin
get; {writeln}
Emit(SelectStdIO,O);
if LexSym.token=LeftParl then begin
get; ()
ScanExpression(ExprRslt);
if ExprRslt.ExprType=FileType then
FlagError(221);
if ExprRslt.ExprType=TextType then begin
Emit(ReleaseIO,O);
Emit(SelectIO,O)
end else begin
if not(ExprRslt.ExprType in [LongintType,RealType,
CharType,StringType]) then FlagError(222);
if LexSym.token=colon1 then begin
get;{}
ScanExpression(FormatExprRslt);
if FormatExprRslt.ExprType <> LongintType then
  FlagError(223);
end else Emit(IntConstant,-1);
if ExprRslt.ExprType=RealType then begin
  if LexSym.token=colon1 then begin
    get;{}
    ScanExpression(FormatExprRslt);
    if FormatExprRslt.ExprType <> LongintType then
      FlagError(223);
  end else Emit(IntConstant,-1);
end;
case ExprRslt.ExprType of
  LongintType : Emit(PrintInt,0);
  RealType : Emit(PrintReal,0);
  CharType : Emit(PrintChar,0);
  StringType : Emit(PrintString,0);
end;
end;
while LexSym.token=comma1 do begin
  get;{}
  ScanExpression(ExprRslt);
  if not(ExprRslt.ExprType in [LongintType,RealType,
  CharType,StringType]) then FlagError(222);
  if LexSym.token=colon1 then begin
    get;{}
    ScanExpression(FormatExprRslt);
    if FormatExprRslt.ExprType <> LongintType then
      FlagError(223);
  end else Emit(IntConstant,-1);
  if ExprRslt.ExprType=RealType then begin
    if LexSym.token=colon1 then begin
      get;{}
      ScanExpression(FormatExprRslt);
      if FormatExprRslt.ExprType <> LongintType then
        FlagError(223);
    end else Emit(IntConstant,-1);
  end;
case ExprRslt.ExprType of
  LongintType : Emit(PrintInt,0);
  RealType : Emit(PrintReal,0);
  CharType : Emit(PrintChar,0);
  StringType : Emit(PrintString,0);
end;
end;
expect(RightPar1,4);
Emit(PrintEol,0);
Emit(FlushIO,0);
Emit(ReleaseIO,0);
end;
    else FlagError(-5);
end;
end;

procedure ScanIncStat;
var TypePtr : integer; Islnc : boolean;
    ExprRslt : ExprDataType;
begin
    Islnc := LexSym.token=incl;
    get; {inc}
    expect(LeftParl,9);
    ScanVarAddress (TypePtr,true,false);
    if not (TypeTabl^[TypePtr].TypeDesc in [IntType,
        LongintType,ShortintType,ByteType,WordType,SubrangeType])
    then FlagError(227);
    if LexSym.token = comma1 then begin
        get; { , }
        ScanExpression(ExprRslt);
        if ExprRslt.ExprType <> LongintType then FlagError(228);
    end else Emit(IntConstant,l);
    expect(RightParl,4);
    if Islnc then Emit(IncVar,TypePtr)
        else Emit(DecVar,TypePtr);
end;

procedure ScanSystemStat;
var ExprRslt : ExprDataType;
    dummy : integer;
    TmpToken : LexOutput;
    done,
    found,
    DoLong : boolean;
    IdData : IdRec;
begin
    case LexSym.token of
        Inline1 : begin
            get; {inline}
            expect(LeftParl,9);
            repeat
                DoLong := LexSym.token = GreaterThen1;
                if DoLong then get;
                if not (LexSym.token in [LongintConst1,identifier1])
                then FlagError(61);
                if LexSym.token=LongintConst1 then begin
                    if DoLong then Emit(DecWord,(LexSym.i1 shr 16) and $ffff);
                    Emit(DecWord,LexSym.i1 AND $ffff);
                end else begin
                    GetIdDesc(LexSym.i1,IdData,found);
                    if not found then FlagError(104);
                    if IdData.IdTyp <> LongintConst then FlagError(61);
if DoLong then Emit(DecWord,(IdData.valptr shr 16) and $ffff);
  Emit(DecWord,IdData.valptr AND $ffff);
end;
get;
done := LexSym.token <> divide1;
if not done then get;
until done;
expect(RightParl,4);
end;

SetExceptionVec1 : begin
  if not RomCode then FlagError(286);
  get;
  expect(LeftParl,9);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> LongintType then
    FlagError(219);
  expect(comma1,20);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> PtrType then
    FlagError(320);
  Emit(StoreVector,0);
  expect(RightParl,4);
end;

Movel : begin
  get;
  expect(LeftParl,9);
  ScanVarAddress(dummy,false,false);
  expect(comma1,20);
  ScanVarAddress(dummy,true,false);
  expect(comma1,20);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> LongintType then
    FlagError(219);
  Emit(DoMove,0);
  expect(RightParl,4);
end;

FillChar1 : begin
  get;
  expect(LeftParl,9);
  ScanVarAddress(dummy,true,false);
  expect(comma1,20);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> LongintType then
    FlagError(219);
  expect(comma1,20);
  ScanExpression(ExprRslt);
  if ExprRslt.ExprType <> LongintType then
    FlagError(219);
  Emit(DoFillChar,0);
  expect(RightParl,4);
end;
procedure ScanSharedVarStat;
var ShrPtr, TempPtr, TypePtr : integer;
   DoRead : boolean;
   ij : word;
begin
  DoRead := LexSym.token=RequestRead1;
case LexSym.token of
    RequestRead1, RequestWrite1 : begin
      get;
      expect(LeftParl,9);
      ScanVarAddress(ShrPtr,false,true);
      if TypeTabl^[ShrPtr].TypeDesc <> SharedType then
        FlagError(402);
      expect(comma1,20);
      ScanVarAddress(TempPtr,false,true);
      if TypeTabl^[TempPtr].TypeDesc <> TemplateType then
        FlagError(403);
      if TypeTabl^[ShrPtr].SubType <> TypeTabl^[TempPtr].SubType then
        FlagError(404);
      if DoRead then Emit(ReqRead,TempPtr)
        else Emit(ReqWrite,TempPtr);
      TypePtr := TypeTabl^[TempPtr].SubType;
      for i := 1 to ArrayDepth(TypePtr) do begin
        expect комma1,20);
        if LexSym.token=times1 then begin
          get;
          Emit(IntConstant,
            TypeTabl^[TypeTabl^[TypePtr].p1].p1);
          Emit(IntConstant,
            TypeTabl^[TypeTabl^[TypePtr].p1].p2);
        end else for j := 1 to 2 do begin
          ScanArrayIndex(TypePtr);
          if j=1 then expect(DotDot1,22);
          end;
        Emit(TempRangeStore,i);
        TypePtr := TypeTabl^[TypePtr].p2;
        end;
        expect(RightParl,4);
        Emit(FinishTemplateOp,TempPtr);
      end;
      Release1 : begin
        get;
        expect(LeftParl,9);
        ScanVarAddress(TempPtr,false,true);
        if TypeTabl^[TempPtr].TypeDesc <> TemplateType then
          FlagError(403);
        expect(RightParl,4);
Emit(TempRelease, TempPtr);
end;
end;

procedure ScanExitStat;
begin
  Emit(FixStack, 0);
  Emit(JmpAlways, ExitLabel);
  get;
end;

procedure AddFieldsToIDTabl(RecTypePtr: integer);
var
  i: longint;
begin
  inc(WithStackPtr);
  with TypeTabl[RecTypePtr] do
    for i := p1 to p1 + p2 - 1 do begin
      with FieldTabl[i] do
        DeclareVar(FName, 0); { use DeclareVar to check for ID table ovrlw}
        IdTabl[IdTablPtr].level := WithStackPtr; { with indicator }
        IdTabl[IdTablPtr].valptr := i; { field table index }
        IdTabl[IdTablPtr].IdTyp := WithField; { mark as with field }
      end;
  end;
end;

procedure ScanWithStat;
var
  OldIDTablPtr: longint;
  RecTypePtr: integer;
{ FuncCall : boolean; }
  WithCount: word;
begin
  get; { WITH }
  OldIDTablPtr := IdTablPtr; { save this }
  ScanVarAddress(RecTypePtr, {FuncCall, true, false});
  if TypeTabl[RecTypePtr].TypeDesc <> RecordType then
    FlagError(140);
  AddFieldsToIDTabl(RecTypePtr);
  WithCount := 1;
  while LexSym.token = comma1 do begin
    get; {, }
    ScanVarAddress(RecTypePtr, {FuncCall, true, false});
    if TypeTabl[RecTypePtr].TypeDesc <> RecordType then
      FlagError(140);
    AddFieldsToIDTabl(RecTypePtr);
    inc(WithCount);
  end;
  expect(do1_54);
  IdTablPtr := OldIDTablPtr; { restore this }
procedure ScanStatAct;
begin
  Emit(Initialize, 0);
  Emit(Trace, 0);
  case LexSym.token of
    identifier1, ptr1 : ScanSmpStat;
    if1 : ScanIfStat;
    for1 : ScanForStat(false);
    while1 : ScanWhileStat;
    case1 : ScanCaseStat;
    repeat1 : ScanRepeatStat;
    begin1 : ScanCmpdStat;
    fassign1, reset1, rewrite1, close1,
    seek1, read1, readln1, write1, writeln1 : ScanIoStat;
    inc1, dec1 : ScanIncStat;
    SetExceptionVec1,
    Inline1,
    FillChar1,
    Move1 : ScanSystemStat;
    parallel1, select1 : ScanParallelStat;
    RequestRead1,
    RequestWrite1,
    Release1 : ScanSharedVarStat;
    exit1 : ScanExitStat;
    with1 : ScanWithStat;
  end;
end;

begin
  WithStackPtr := 0;
  ScanStat := ScanStatAct;
  ScanCmpdStat := ScanCmpdStatAct;
  ScanVarAddress := ScanVarAddressAct;
  ScanForStat := ScanForStatAct;
end.

unit scontyp;
{used to scan constant and type declarations}

interface
uses
crt,
germsym,  {used to obtain next token}
lex,      {used, if necessary, to obtain error positions}
types,    {declares all types used}
strstore,  {used to store and retrieve string constants}
options,   {used to flag all options - call to check options}
tables,    {stores all compiler created tables}
CircRef, \{used for procedural variables\}
errors; \{used to flag all errors\}

procedure ScanConstExprfvar ExprResult:ConstResultType);
procedure ScanConstDecAct;
procedure ScanTypeAct(var TypePtr:integer);
procedure ScanTypeDecAct;

implementation

procedure ScanConstExpr(var ExprResult:ConstResultType);

const
  MaxStackLevel = 16;

type
  ConstStackEntryType = ConstResultType;

ConstStackType = array[0..MaxStackLevel] of ConstStackEntryType;

var temp:LexOutput; StackPtr : longint;v1,v2 : longint;
  ConstStack : ConstStackType;
  s1,s2 : DataString;
  Rvl, Rv2 : double;

procedure IncStackPtr;
begin
  inc(StackPtr);
  if StackPtr > MaxStackLevel then FlagError(259);
end;

procedure FindEnum(id:longint;var success:boolean);
var i:longint;
begin
  i := EnumTablPtr;
  success := false;
  while (not success) and (i>0) do
    if EnumTabl^ [i] = id then success := true else dec(i);
end;

procedure ScanConstFactor;
var
  SubExprResult : ConstResultType;
  IdFound : boolean;
  IdData : IdRec;
  SizeOfTPtr : integer;
begin
  case LexSym.token of
    true1 : begin
      IncStackPtr;
      ConstStack[StackPtr].CType := BoolConst;
    end;
end;
ConstStack[StackPtr].ConstantVal := 1;
Get;
end;
false1 : begin
  IncStackPtr;
  ConstStack[StackPtr].CType := BoolConst;
  ConstStack[StackPtr].ConstantVal := 0;
  Get;
end;
StringConst1 : begin
  IncStackPtr;
  ConstStack[StackPtr].CType := StrConst;
  ConstStack[StackPtr].ConstantVal := LexSym.il;
  Get;
end;
LongintConst1 : begin
  IncStackPtr;
  ConstStack[StackPtr].CType := LongintConst;
  ConstStack[StackPtr].ConstantVal := LexSym.il;
  Get;
end;
CharConst1 : begin
  IncStackPtr;
  ConstStack[StackPtr].CType := CharConst;
  ConstStack[StackPtr].ConstantVal := LexSym.il;
  Get;
end;
RealConst1 : begin
  IncStackPtr;
  ConstStack[StackPtr].CType := RealConst;
  ConstStack[StackPtr].ConstantVal := LexSym.il;
  Get;
end;
LeftPar1 : begin
  Get; // get rid of the {} 
  ScanConstExpr(SubExprResult);
  IncStackPtr;
  ConstStack[StackPtr].CType := SubExprResult.CType;
  ConstStack[StackPtr].ConstantVal := SubExprResult.ConstantVal;
  Expect(RightPar1,4);
end;
SizeOf1 : begin
  Get;
  Expect(LeftPar1,9);
  ScanType(SizeOfTPtr);
  Expect(RightPar1,4);
  IncStackPtr;
  ConstStack[StackPtr].CType := LongintConst;
  ConstStack[StackPtr].ConstantVal := TypeTabl^ [SizeOfTPtr].TypeSize;
end;
identifier1 : begin
GetIdDesc(LexSym.il,IdData,IdFound);
if not IdFound then begin
  FindEnum(LexSym.il,IdFound);
  if IdFound then begin
    IncStackPtr;
    ConstStack[StackPtr].CType := EnumConst;
    { note that for enum constants we store no value }
    { we store only the id number of the identifier }
    { this is used by other compiler sections to find value }
    ConstStack[StackPtr].ConstantVal := LexSym.il;
    IdData.IdTyp := EnumConst;
  end;
end;
if not IdFound then FlagError(104);
if (IdData.IdTyp in [BoolConst,
  LongintConst,StrConst,CharConst,RealConst]) then begin
  IncStackPtr;
  ConstStack[StackPtr].CType := IdData.IdTyp;
  ConstStack[StackPtr].ConstantVal := IdData.valptr;
end else if not IdFound then FlagError(50);
Get;
end;
end;

procedure ScanConstTerm;
var temp : LexOutput; R1, R2 : double;
begin
  ScanConstFactor;
  while (LexSym.token in [timesl,dividel,divl,modl,andl]) do begin
    temp := LexSym; {save the term op here)
    Get; {move on}
    ScanConstFactor;
    case temp.token of
      andl : begin
        if (ConstStack[StackPtr].CType=BoolConst) and
          (ConstStack[StackPtr-1].CType=BoolConst) then begin
          if (ConstStack[StackPtr-1].ConstantVal=1) and
            (ConstStack[StackPtr].ConstantVal=1) then
            ConstStack[StackPtr-1].ConstantVal := 1
        end else begin
          ConstStack[StackPtr-1].ConstantVal := 0;
          dec(StackPtr);
        end else begin
          ConstStack[StackPtr-1].ConstantVal := 1
        end else if (ConstStack[StackPtr].CType=LongintConst) and
          (ConstStack[StackPtr-1].CType=LongintConst) then
          begin
            ConstStack[StackPtr-1].ConstantVal :=
            ConstStack[StackPtr-1].ConstantVal AND
            ConstStack[StackPtr].ConstantVal;
          end;
      end;
  end;
end:
dec(StackPtr);
end else FlagError(129);
end;

times1 : begin
if (ConstStack[StackPtr]. CType = LongintConst) and
(ConstStack[StackPtr-1]. CType = LongintConst) then
begin
ConstStack[StackPtr-1]. ConstantVal :=
ConstStack[StackPtr-1]. ConstantVal *
ConstStack[StackPtr]. ConstantVal;
end;
end;

div1 : begin
if (ConstStack[StackPtr]. CType = LongintConst) and
(ConstStack[StackPtr-1]. CType = LongintConst) then
begin
if ConstStack[StackPtr]. ConstantVal <> 0 then
ConstStack[StackPtr-1]. ConstantVal :=
ConstStack[StackPtr-1]. ConstantVal div
ConstStack[StackPtr]. ConstantVal
else FlagError(300);
dec(StackPtr);
end else FlagError(129);
end;
divide1 : begin
if (ConstStack[StackPtr]. CType in
[LongintConst, RealConst]) and
(ConstStack[StackPtr-1]. CType in
[LongintConst, RealConst]) then
begin
if ConstStack[StackPtr-1]. CType = LongintConst then
end;
R1 := ConstStack[StackPtr-1].ConstantVal  
else  
  GetRealConstant(R1,ConstStack[StackPtr-1].ConstantVal);  
if ConstStack[StackPtr].CType = LongintConst then  
  R2 := ConstStack[StackPtr].ConstantVal  
else  
  GetRealConstant(R2,ConstStack[StackPtr].ConstantVal);  
if R2 = 0.0 then FlagError(300);  
dec(StackPtr);  
ConstStack[StackPtr].CType := RealConst;  
SaveRealConstant(R1/R2,ConstStack[StackPtr].ConstantVal);  
end else FlagError(129);  
end;  
mod1 : begin  
if (ConstStack[StackPtr].CType= LongintConst) and  
  (ConstStack[StackPtr-1].CType=LongintConst) then  
begin  
if ConstStack[StackPtr].ConstantVal <> 0 then  
  ConstStack[StackPtr-1].ConstantVal :=  
  ConstStack[StackPtr-1].ConstantVal mod  
  ConstStack[StackPtr].ConstantVal
else FlagError(300);  
dec(StackPtr);  
end else FlagError(129);  
end;  
end;  
end;  
procedure ScanConstSexpr;  
var negate:boolean; temp:LexOutput;TReal,R1,R2 : double;  
begin  
negate := false;  
if (LexSym.token= minus1) then  
  begin  
    negate := true; {indicate the negation to take place}  
    Get; {get rid of the minus sign}  
  end  
else if (LexSym.token=plus1) then Get; {this is a NOP}  
  ScanConstTerm;  
if negate then  
  case ConstStack[StackPtr].CType of  
    LongintConst : ConstStack[StackPtr].ConstantVal :=  
      -ConstStack[StackPtr].ConstantVal;  
    RealConst : begin  
      GetRealConstant(TReal,ConstStack[StackPtr].ConstantVal);  
      SaveRealConstant(-TReal,ConstStack[StackPtr].ConstantVal);  
    end;  
  else FlagError(129);  
end;
while (LexSym.token in [plusl,minusl,orl]) do
begin
  temp := LexSym;  {save the sexpr op}
  Get;  {move on}
  ScanConstTerm;
  case temp.token of
  orl : begin
    if (ConstStack[StackPtr]. CType = BoolConst)
      and (ConstStack[StackPtr-1]. CType = BoolConst) then begin
      if (ConstStack[StackPtr]. ConstantVal = 1) or
        (ConstStack[StackPtr-1]. ConstantVal = 1)
      then ConstStack[StackPtr-1]. ConstantVal := 1
      else ConstStack[StackPtr-1]. ConstantVal := 0;
      dec (StackPtr);
    end else begin
      if (ConstStack[StackPtr]. CType = LongintConst)
        and (ConstStack[StackPtr-1]. CType = LongintConst) then begin
        ConstStack[StackPtr-1]. ConstantVal :=
        ConstStack[StackPtr-1]. ConstantVal OR
        ConstStack[StackPtr]. ConstantVal;
        dec (StackPtr);
      end else FlagError(129);
    end;
  end;
  minusl : begin
    if (ConstStack[StackPtr]. CType = LongintConst)
      and (ConstStack[StackPtr-1]. CType = LongintConst) then begin
      ConstStack[StackPtr-1]. ConstantVal :=
      ConstStack[StackPtr-1]. ConstantVal -
      ConstStack[StackPtr]. ConstantVal;
      dec (StackPtr);
    end else begin
      if (ConstStack[StackPtr]. CType in
        [LongintConst, RealConst]) and
        (ConstStack[StackPtr-1]. CType in
        [LongintConst, RealConst]) then begin
        if ConstStack[StackPtr-1]. CType = LongintConst then
          R1 := ConstStack[StackPtr-1]. ConstantVal
        else
          GetRealConstant(R1, ConstStack[StackPtr-1]. ConstantVal);
        if ConstStack[StackPtr]. CType = LongintConst then
          R2 := ConstStack[StackPtr]. ConstantVal
        else
          GetRealConstant(R2, ConstStack[StackPtr]. ConstantVal);
        dec (StackPtr);
        ConstStack[StackPtr]. CType := RealConst;
        SaveRealConstant(R1-R2, ConstStack[StackPtr]. ConstantVal);
    end;
  end;
end;
end else FlagError(129);
end;
end;

plus1 : begin
if (ConstStack[StackPtr].CType=LongintConst)
and (ConstStack[StackPtr-1].CType=LongintConst) then
begin
ConstStack[StackPtr-1].ConstantVal :=
    ConstStack[StackPtr-1].ConstantVal +
    ConstStack[StackPtr].ConstantVal;
dec(StackPtr);
end else begin
if (ConstStack[StackPtr].CType in [CharConst,StrConst])
and (ConstStack[StackPtr-1].CType in [CharConst,StrConst])
then
begin
    if ConstStack[StackPtr-1].CType=CharConst
    then s1 := chr(ConstStack[StackPtr-1].ConstantVal)
    else GetStringConstant(s1,ConstStack[StackPtr-1].ConstantVal);
    if ConstStack[StackPtr].CType=CharConst
    then s2 := chr(ConstStack[StackPtr].ConstantVal)
    else GetStringConstant(s2,ConstStack[StackPtr].ConstantVal);
    s1 := s1 + s2;
    dec(StackPtr);
    ConstStack[StackPtr].CType := StrConst;
    SaveStringConstant(s1,ConstStack[StackPtr].ConstantVal);
end else begin
    if (ConstStack[StackPtr].CType in
        [LongintConst,RealConst]) and
    (ConstStack[StackPtr-1].CType in
        [LongintConst,RealConst]) then
    begin
        if ConstStack[StackPtr-1].CType = LongintConst then
            R1 := ConstStack[StackPtr-1].ConstantVal
        else
            GetRealConstant(R1,ConstStack[StackPtr-1].ConstantVal);
        if ConstStack[StackPtr].CType = LongintConst then
            R2 := ConstStack[StackPtr].ConstantVal
        else
            GetRealConstant(R2,ConstStack[StackPtr].ConstantVal);
        dec(StackPtr);
        ConstStack[StackPtr].CType := RealConst;
        SaveRealConstant(R1+R2,ConstStack[StackPtr].ConstantVal);
    end else FlagError(129);
end;
end;
begin {ScanConstExpr}
StackPtr := -1; {initialize the push-down stack pointer}
ScanConstSexpr;
if (LexSym.token in [equal1,NotEqual1,LessThen1,
  GreaterThen1,LessEq1,GreaterEq1]) then
begin
  temp := LexSym; {save this temporarily}
  Get; {get the next token}
  ScanConstSexpr;
  if (ConstStack[0].CType <> ConstStack[1].CType) then
  begin
    if (ConstStack[0].CType in [LongintConst,RealConst]) and
        (ConstStack[1].CType in [LongintConst,RealConst]) then
    begin
      if ConstStack[0].CType=LongintConst then
      begin
        Rv1 := ConstStack[0].ConstantVal;
        SaveRealConstant(Rv1,ConstStack[0].ConstantVal);
        ConstStack[0].CType := RealConst;
      end;
      if ConstStack[1].CType=LongintConst then
      begin
        Rv1 := ConstStack[1].ConstantVal;
        SaveRealConstant(Rv1,ConstStack[1].ConstantVal);
        ConstStack[1].CType := RealConst;
      end;
    end else FlagError(129);
  end;
  case ConstStack[0].CType of
  LongintConst,CharConst : begin
    v1 := ConstStack[0].ConstantVal;
    v2 := ConstStack[1].ConstantVal;
    ConstStack[0].CType := BoolConst;
    ConstStack[0].ConstantVal := 0; {0 = false, 1 = true}
    case temp.token of
    equal1 : if v1=v2 then ConstStack[0].ConstantVal := 1;
    NotEqual1 : if v1 <> v2 then ConstStack[0].ConstantVal := 1;
    GreaterThen1 : if v1 > v2 then ConstStack[0].ConstantVal := 1;
    LessThen1 : if v1 < v2 then ConstStack[0].ConstantVal := 1;
    LessEq1 : if v1 <= v2 then ConstStack[0].ConstantVal := 1;
    GreaterEq1 : if v1 >= v2 then ConstStack[0].ConstantVal := 1;
    end;
    dec(StackPtr);
  end;
  StrConst : begin
    GetStringConstant(s1,ConstStack[0].ConstantVal);
    GetStringConstant(s2,ConstStack[1].ConstantVal);
    ConstStack[0].CType := BoolConst;
    ConstStack[0].ConstantVal := 0; {0 = false, 1 = true}
    case temp.token of
    equal1 : if s1=s2 then ConstStack[0].ConstantVal := 1;
end
end;
end;
NotEqual1 : if s1 <> s2 then ConstStack[0].ConstantVal := 1;
GreaterThen1 : if s1 > s2 then ConstStack[0].ConstantVal := 1;
LessThen1 : if s1 < s2 then ConstStack[0].ConstantVal := 1;
LessEq1 : if s1 <= s2 then ConstStack[0].ConstantVal := 1;
GreaterEq1 : if s1 >= s2 then ConstStack[0].ConstantVal := 1;
end;
dec(StackPtr);
end;

RealConst: begin
GetRealConstant(Rv1,ConstStack[0].ConstantVal);
GetRealConstant(Rv2,ConstStack[1].ConstantVal);
ConstStack[0].CType := BoolConst;
ConstStack[0].ConstantVal := 0; {0 = false, 1 = true}
case temp.token of
  equal1 : if Rv1 = Rv2 then ConstStack[0].ConstantVal := 1;
  NotEqual1 : if Rv1 <> Rv2 then ConstStack[0].ConstantVal := 1;
  GreaterThen1 : if Rv1 > Rv2 then ConstStack[0].ConstantVal := 1;
  LessThen1 : if Rv1 < Rv2 then ConstStack[0].ConstantVal := 1;
  LessEq1 : if Rv1 <= Rv2 then ConstStack[0].ConstantVal := 1;
  GreaterEq1 : if Rv1 >= Rv2 then ConstStack[0].ConstantVal := 1;
end;
dec(StackPtr);
end;
ExprResult := ConstStack[StackPtr];
end;

procedure ScanConstDecAct;
var IdentNum : longint; ConstResult : ConstResultType;
begin
Get; {clear out the CONST}
if LexSym.token <> identifier1 then FlagError(2);
while (LexSym.token=identifier1) do
begin
  IdentNum := LexSym.il; {save the identifier number}
  {check if this name is already in use}
  if AlreadyDef(IdentNum) then FlagError(101)
  else Get; {clear out the identifier}
  Expect(equal1,16);
  ScanConstExpr(ConstResult);
  case ConstResult.CType of
    LongintConst : DecConst(LongintConst,IdentNum,ConstResult.ConstantVal);
    RealConst : DecConst(RealConst,IdentNum,ConstResult.ConstantVal);
    CharConst : DecConst(CharConst,IdentNum,ConstResult.ConstantVal);
    StrConst : DecConst(StrConst,IdentNum,ConstResult.ConstantVal);
    BoolConst : DecConst(BoolConst,IdentNum,ConstResult.ConstantVal);
    EnumConst : DecConst(EnumConst,IdentNum,ConstResult.ConstantVal);
    else FlagError(50);
procedure ScanTypeAct(var TypePtr:integer);
var IdFound : boolean;IdResult:IdRec;

procedure ScanArrayType(var NtypePtr:integer);
var size : longint;ElementPtr,RangePtr:integer;
  row,col,fnum : integer;
begin
Get; \{comma or [\}
GetStatus(row,col,fnum);
ScanType(RangePtr);
if TypeTabl^ [RangePtr].TypeDesc=FwdPtrType then
  FlagErrorPos(118,row,col,fnum);
if not(TypeTabl^ [RangePtr].TypeDesc in [IntType,
   LongintType,ShortintType,ByteType,WordType,CharType,BoolType,
   SubrangeType,EnumType]) then FlagError(113);
if (LexSym.token = RightBrktl) then
begin
Get; \{get rid of the ]\}
if (LexSym.token=LeftBrktl) then
  ScanArrayType(ElementPtr)
else begin
  Expect(ofl,8);
  GetStatus(row,col,fnum);
  ScanType(ElementPtr);
  if TypeTabl^ [ElementPtr].TypeDesc=FwdPtrType then
    FlagErrorPos(118,row,col,fnum);
end;
end else begin
if (LexSym.token=comma1) then
  ScanArrayType(ElementPtr)
else FlagError(12);
end;
case TypeTabl^ [RangePtr].TypeDesc of
  IntType,WordType : size := 65536;
  LongintType : FlagError(187);
  CharType,ShortintType,ByteType,WordType,CharType,BoolType,
  SubrangeType,EnumType : size := TypeTabl^ [RangePtr].p2;
  else FlagError(113);
end;
size := size * TypeTabl^ [ElementPtr].TypeSize;
if size > MaxTypeSize then FlagError(187);
StoreTypeData(ArrayType,0,RangePtr,ElementPtr,size,
PROCEDURE ScanArrayType;
VAR
 size : LongInt; RangePtr, ElementPtr : Integer;
    row, col, fnum : Integer;
BEGIN
   Get; {ARRAY}
   Expect(LeftBrkt1, 11);
   GetStatus(row, col, fnum);
   ScanType(RangePtr);
   IF TypeTabl^ [RangePtr].TypeDesc = FwdPtrType THEN
      FlagErrorPos(118, row, col, fnum);
   IF NOT(TypeTabl^ [RangePtr].TypeDesc in [IntType, LongintType, ShortintType, ByteType, WordType, CharType, BoolType, SubrangeType, EnumType]) THEN FlagError(113);
   IF (LexSym.token = RightBrkt1) THEN
      BEGIN
         Get; {get rid of the }
         IF (LexSym.token = LeftBrkt1) THEN
            ScanNarrayType(ElementPtr)
         ELSE BEGIN
            Expect(of1, 8);
            GetStatus(row, col, fnum);
            ScanType(ElementPtr);
            IF TypeTabl^ [ElementPtr].TypeDesc = FwdPtrType THEN
               FlagErrorPos(118, row, col, fnum);
            END;
         END ELSE BEGIN
            IF (LexSym.token = comma1) THEN
               ScanNarrayType(ElementPtr)
            ELSE FlagError(12);
         END;
      END;
   END;
   size := 1;
   CASE TypeTabl^ [RangePtr].TypeDesc OF
      IntType, WordType, LongintType,
      CharType, ShortintType, ByteType,
      BoolType, SubrangeType :
         size := TypeTabl^ [RangePtr].p2 - TypeTabl^ [RangePtr].p1 + 1;
      EnumType : size := TypeTabl^ [RangePtr].p2;
      ELSE FlagError(113);
   END;
   IF (MaxTypeSize div size) < TypeTabl^ [ElementPtr].TypeSize THEN
      FlagError(187);
   size := size * TypeTabl^ [ElementPtr].TypeSize;
   IF size > MaxTypeSize THEN FlagError(187);
   StoreTypeData(ArrayType, 0, RangePtr, ElementPtr, size,
                  TypeTabl^ [ElementPtr].AddrMask);
   TypePtr := TypeTablPtr;
END;
procedure ScanSetType;
var row,col,fnum:integer;
function SetSize(TypePtr:longint):longint;
var s1,s2:longint;
begin
  s1 := TypeTabl^[TypePtr].p2;
s2 := TypeTabl^[TypePtr].p1;
s1 := 1 + s1 - s2;
  if (s1 mod 8) = 0 then SetSize := s1 div 8
  else SetSize := (s1 div 8) + 1;
end;

begin
  Get; {clear out the SET}
  Expect(of1,8);
  GetStatus(row,col,fnum);
  ScanType(TypePtr);
  if TypeTabl^[TypePtr].TypeDesc=FwdPtrType then
    FlagErrorPos(118,row,col,fnum);
  if not (TypeTabl^[TypePtr].TypeDesc in [CharType,BoolType,SubrangeType,
    EnumType]) then FlagError(136);
  if SetSize(TypePtr) > 32 then FlagError(182)
  else
    StoreTypeData(SetType,TypePtr,0,0,SetSize(TypePtr),
      TypeTabl^[TypePtr].AddrMask);
  TypePtr := TypeTablPtr;
end;

procedure ScanEnumType;
var count : integer; ptr : integer; check:integer;
begin
  count := 0; ptr := EnumTablPtr;
  Get; {get rid of the ()}
  if (LexSym.token<>identifier1) then FlagError(2);
  EnumTabl^[ptr] := LexSym.i1;
  inc(count); inc(ptr);
  Get;
  while (LexSym.token=comma1) do
    begin
      Get; {clear out the comma}
      if (LexSym.token=identifier1) then
        begin
          EnumTabl^[ptr] := LexSym.i1;
          for check := EnumTablPtr to ptr-1 do
            if EnumTabl^[check]=LexSym.i1 then FlagError(184);
          inc(count); inc(ptr);
          if ptr >= MaxNumEnum then FlagError(183);
          Get;
        end;
    end;
  if count>256 then FlagError(183);
procedure ScanRecordType;
var size,count,icount,TmpTablPtr:longint;
   FieldTypePtr:integer;
   IdNums : array[1..64] of FieldDescType;
   check,i : integer;
   row,col,fnum:integer;
   RecordAddrMask : longint;
begin
Get; {RECORD}
size := 0;
count := 0;
RecordAddrMask := 0;
while (LexSym.token<>endl) do
begin
   icount := 0;
   if (LexSym.token=identifierl) then
   begin
      inc (icount);
      if (icount+count)>64 then begin FlagError(263); dec(icount); end;
      IdNums[icount+count].FName := LexSym.il;
      for check := count+icount-1 downto 1 do
         if IdNums[check].FName=LexSym.il then FlagError(lOl);
      Get;
   end else FlagError(2);
   while (LexSym.token=comma) do begin
      Get; {comma}
      if (LexSym.token=identifierl) then
      begin
         inc(icount);
         if (icount+count)>64 then begin FlagError(263); dec(icount); end;
         IdNums[icount+count].FName := LexSym.il;
         for check := count+icount-1 downto 1 do
            if IdNums[check].FName=LexSym.il then FlagError(101);
         Get;
      end else FlagError(2);
   end;
   Expect(colonl,5);
GetStatus(row,col,fnum);
ScanType(FieldTypePtr);
RecordAddrMask := RecordAddrMask OR
   [FieldTypePtr].AddrMask;
if FieldTypePtr^.TypeDesc=FwdPtrType then
   FlagErrorPos(118,row,col,fnum);
Expect(semicolonl, 14);
for i := count+1 to count+icount do
begin
  with IdNums[i] do
  begin
    FTypePtr := FieldTypePtr;
    AdjustAddr(size,TypeTabl^[FieldTypePtr].AddrMask);
    AddrOffs := size;
    end;
    size := size + TypeTabl^[FieldTypePtr].TypeSize;
    size := (size + 1) AND $fffffffe;
  end;
  count := count + icount;
end;
if FieldTablPtr+count > MaxNumFields then FlagError(264);
for i := 1 to count do
  FieldTabl^[FieldTablPtr+i] := IdNums[i];
  StoreTypeData(RecordType,0,FieldTablPtr+l,
    count,size,RecordAddrMask);
  TypePtr := TypeTablPtr;
  FieldTablPtr := FieldTablPtr + count;
  Expect(endl,13);
end;

procedure ScanFileType;
var ElementPtr: integer;
  row,col,fnum: integer;
begin
  Get; {FILE}
  Expect(ofl,8);
  GetStatus(row,col,fnum);
  ScanType(ElementPtr);
  if TypeTabl^[ElementPtr].TypeDesc=FwdPtrType then
    FlagErrorPos(118,row,col,fnum);
  if ContainsType(PtrType,ElementPtr,0) then
    FlagErrorPos(312,row,col,fnum);
  if ContainsType(FileType,ElementPtr,0) then
    FlagErrorPos(313,row,col,fnum);
  if ContainsType(TextType,ElementPtr,0) then
    FlagErrorPos(313,row,col,fnum);
  if ContainsType(ResourceType,ElementPtr,0) then
    FlagErrorPos(314,row,col,fnum);
  if ContainsType(ProcessType,ElementPtr,0) then
    FlagErrorPos(315,row,col,fnum);
  if ContainsType(ChannelType,ElementPtr,0) then
    FlagErrorPos(316,row,col,fnum);
  StoreTypeData(FileType,ElementPtr,0,0,90,3);{90 bytes in size}
  TypePtr := TypeTablPtr;
end;

procedure ScanChannelType;
var ElementPtr: integer;
  row,col,fnum: integer;
begin
Get; {channel}
Expect(of1,8);
GetStatus(row,col,fnum);
ScanType(ElementPtr);
if TypeTabl^[ElementPtr].TypeDesc=FwdPtrType then
  FlagErrorPos(118,row,col,fnum);
if ContainsType(FileType,ElementPtr,0) then
  FlagErrorPos(313,row,col,fnum);
if ContainsType(TextType,ElementPtr,0) then
  FlagErrorPos(313,row,col,fnum);
if ContainsType(PtrType,ElementPtr,0) then
  FlagErrorPos(312,row,col,fnum);
if ContainsType(ResourceType,ElementPtr,0) then
  FlagErrorPos(314,row,col,fnum);
if ContainsType(ProcessType,ElementPtr,0) then
  FlagErrorPos(315,row,col,fnum);
if ContainsType(ChannelType,ElementPtr,0) then
  FlagErrorPos(316,row,col,fnum);
StoreTypeData(ChannelType,ElementPtr,0,0,4,1);{4 bytes in size}
TypePtr := TypeTablPtr;
end;

procedure ScanPtrType;
var
  ElementPtr: integer;
  row,col,FileStatNum:integer;
  exists:boolean;
  IdData:IdRec;
begin
  Get; {^}
  if LexSym.token=identifier1 then
    GetIdDesc(LexSym.i1,IdData,exists)
  else exists := false;
  if (LexSym.token=identifier1) and (not exists) then begin
    GetStatus(row,col,FileStatNum);
    StoreTypeData(FwdPtrType,row,col,FileStatNum,0,3);
    ElementPtr := TypeTablPtr;
    DecType(LexSym.i1,ElementPtr);
    StoreTypeData(PtrType,ElementPtr,0,0,4,3);{4 bytes in size}
    TypePtr := TypeTablPtr;
    Get;
  end else begin
    ScanType(ElementPtr);
    StoreTypeData(PtrType,ElementPtr,0,0,4,3);{4 bytes in size}
    TypePtr := TypeTablPtr;
  end;
end;

procedure ScanSharedType;
var ElementPtr : integer;
Depth : longint;
row, col, fnum : integer;

begin
Get;
GetStatus(row, col, fnum);
ScanType(ElementPtr);
if TypeTabl^[ElementPtr].TypeDesc=FwdPtrType then
  FlagErrorPos(118, row, col, fnum);
if ContainsType(PtrType, ElementPtr, 0) then
  FlagErrorPos(312, row, col, fnum);
if ContainsType(ResourceType, ElementPtr, 0) then
  FlagErrorPos(314, row, col, fnum);
if ContainsType(ProcessType, ElementPtr, 0) then
  FlagErrorPos(315, row, col, fnum);
if ContainsType(ChannelType, ElementPtr, 0) then
  FlagErrorPos(316, row, col, fnum);
Depth := ArrayDepth(ElementPtr);
StoreTypeData(SharedType, ElementPtr, Depth, 0, 4, 3);
{size = 4 : contains physical address of shared var control page}
TypePtr := TypeTablPtr;
end;

procedure ScanTemplateType;
var ElementPtr : integer;
  Depth : longint;
  row, col, fnum : integer;

begin
Get;
Expect(ofl, 8);
GetStatus(row, col, fnum);
ScanType(ElementPtr);
if TypeTabl^[ElementPtr].TypeDesc=FwdPtrType then
  FlagErrorPos(118, row, col, fnum);
if ContainsType(PtrType, ElementPtr, 0) then
  FlagErrorPos(312, row, col, fnum);
if ContainsType(ResourceType, ElementPtr, 0) then
  FlagErrorPos(314, row, col, fnum);
if ContainsType(ProcessType, ElementPtr, 0) then
  FlagErrorPos(315, row, col, fnum);
if ContainsType(ChannelType, ElementPtr, 0) then
  FlagErrorPos(316, row, col, fnum);
Depth := ArrayDepth(ElementPtr);
StoreTypeData(TemplateType, ElementPtr, Depth, 0, 16 + 8 * Depth, 1);
{size =
  2 bytes for status, $00
  2 bytes for template identifier, $02
  4 bytes for template process ID, $04
  4 bytes for shared variable control page, $08
  4 bytes for address of shared variable, $0c
  8 bytes per array dimension $10
}
procedure ScanStandardType;
var temp:TypeDescType;
begin
  case LexSym.token of
    integerl : temp := IntType;
    longintl : temp := LongintType;
    shortintl : temp := ShortintType;
    byte1 : temp := ByteType;
    word1 : temp := WordType;
    real1 : temp := RealType;
    double1 : temp := DoubleType;
    char1 : temp := CharType;
    boolean1 : temp := BoolType;
    text1 : temp := TextType;
    stringl : temp := StringType;
    pointer1 : begin
      TypePtr := NilPointerTypePtr;
      Get;
      Exit;
    end;
  end;
  TypePtr := 1;
  while TypeTabl^[TypePtr].TypeDesc <> temp do inc(TypePtr);
  Get;
end;

procedure ScanSubrangeType;
var CRsltl,CRslt2 : ConstResultType;

procedure StoreEnumSubrange;
var i,j:integer;found,found1,found2:boolean;ord1,ord2:integer;
begin
  i := TypeTablPtr;
  found := false;
  while (not found) and (i>0) do
    begin
      if TypeTabl^[i].TypeDesc = EnumType then
        begin
          found1 := false;
          j := TypeTabl^[i].p1; {pointer to first enum of this type}
          while (not found1) and (j <= TypeTabl^[i].p1 + TypeTabl^[i].p2 - 1) do
            if EnumTabl^[j]=CRsltl.ConstantVal then found1 := true
            else inc(j);
          ord1 := j - TypeTabl^[i].p1;
          found2 := false;
          j := TypeTabl^[i].p1; {pointer to first enum of this type}
          while (not found2) and (j <= TypeTabl^[i].p1 + TypeTabl^[i].p2 - 1) do
            if EnumTabl^[j]=CRslt2.ConstantVal then found2 := true
            else inc(j);
        end;
    end;
end;
else inc(j);
ord2 := j - TypeTabl^[i].p1;
found := found1 and found2;
end;
if found then
  if (ord1 > ord2) then
    FlagError(188);
  if found then StoreTypeData(SubrangeType,i,ord1,ord2,1,
    TypeTabl^[i].AddrMask);
dec(i);
end;
if not found then FlagError(107);
end; {StoreEnumSubrange}

procedure StoreLongintSubrange;
var v1,v2 : longint;TypeSize:integer;
ItypePtr : integer;
begin
  v1 := CRsltl.ConstantVal; v2 := CRslt2.ConstantVal;
  TypeSize := 4; ItypePtr := LongintTypePtr;
  if (v1>=-32768) and (v2<=32767) then
    begin
      TypeSize := 2; ItypePtr := IntTypePtr;
    end;
  if (v1>=0) and (v2<=65535) then
    begin
      TypeSize := 2; ItypePtr := WordTypePtr;
    end;
  if (v1>=-128) and (v2<=127) then
    begin
      TypeSize := 1; ItypePtr := ShortintTypePtr;
    end;
  if (v1>=0) and (v2<=255) then
    begin
      TypeSize := 1; ItypePtr := ByteTypePtr;
    end;
  StoreTypeData(SubrangeType,ItypePtr,CRsltl.ConstantVal,CRslt2.ConstantVal,
    TypeSize,TypeTabl^[ItypePtr].AddrMask);
end;

begin (ScanSubrangeType)
ScanConstExpr(CRsltl);
Expect(DotDot1,22);
ScanConstExpr(CRslt2);
if CRsltl.CType <> CRslt2.CType then FlagError(107);
if not(CRsltl.CType in [LongintConst,CharConst,
  BoolConst,EnumConst])
  then FlagError(186);
if (CRslt2.ConstantVal < CRsltl.ConstantVal)
  and (CRsltl.CType<>EnumConst) then
  FlagError(188);
case CRslt1.CType of
    LongintConst : StoreLongintSubrange;
    CharConst : StoreTypeData(SubrangeType,CharTypePtr,CRslt1.ConstantVal,
                               CRslt2.ConstantVal,1,0);
    BoolConst : StoreTypeData(SubrangeType,BoolTypePtr,CRslt1.ConstantVal,
                               CRslt2.ConstantVal,1,0);
    EnumConst : StoreEnumSubrange;
end;
TypePtr := TypeTablPtr;
end;

begin {ScanType}
    case LexSym.token of
      integer1,longint1,shortint1,
      byte1,word1,char1,boolean1,
      real1,double1,text1,string1,pointer1 :  ScanStandardType;
      array1 :  ScanArrayType;
      set1 :  ScanSetType;
      LeftPar1 :  ScanEnumType;
      record1 :  ScanRecordType;
      resource1 :  ScanResourceType(TypePtr);
      process1 :  ScanProcessType(TypePtr);
      file1 :  ScanFileType;
      channel1 :  ScanChannelType;
      carrot1 :  ScanPtrType;
      LongintConst1,CharConst1,true1,false1,minus1,plus1:ScanSubrangeType;
      shared1 :  ScanSharedType;
      template1 :  ScanTemplateType;
      identifier1 : begin
        GetIdDesc(LexSym.i1,IdResult,IdFound);
        if not IdFound then ScanSubrangeType
        else begin
          case IdResult.IdTyp of
            LongintConst,CharConst,
            BoolConst : ScanSubrangeType;
            TypeName : begin
              TypePtr := IdResult.valptr;
              Get;
            end;
            else FlagError(10);
          end;
        end;
        end;
        end; /* identifier1 */
      procedure ScanTypeDecAct;
      var IdentNum:longint;
      TypePtr : integer;
TypChk:IdRec;
found:boolean;
row,col,fnum:integer;
FwdPointer: integer;
i : integer;
begin
Get; {clear out the TYPE}
if LexSym.token <> identifier1 then FlagError(2);
while LexSym.token=identifier1 do
begin
IdentNum := LexSym.i1; {save the id num}
if AlreadyDef(IdentNum) then
begin
GetIdDesc(IdentNum,TypChk,found);
if not((TypChk.IdTyp=TypeName) and (TypeTabl ^ [TypChk.valptr].
    TypeDesc=FwdPtrType)) then FlagError(101) {is it unique?}
end;
Get; {get rid of the identifier}
Expect(equal1,16); {and get rid of the = }
GetStatus(row,col,fnum); {save this location in case of a FwdPtr type}
ScanType(TypePtr); {scan the type declaration}
if TypeTabl ^ [TypePtr].TypeDesc=FwdPtrType then
    FlagErrorPos(118,row,col,fnum); {don't allow forward pointer types}
if AlreadyDef(IdentNum) then
begin
GetIdDesc(IdentNum,TypChk,found); {found will be true}
if (TypChk.IdTyp=TypeName) and (TypeTabl ^ [TypChk.valptr].
    TypeDesc=FwdPtrType) then
begin
    FwdPointer := TypChk.valptr; {get the entry number of the FwdPtr}
    for i := TypeTablPtr downto 1 do
        if (TypeTabl ^ [i].TypeDesc = PtrType) and
            (TypeTabl ^ [i].SubType = FwdPointer) then
            TypeTabl ^ [i].SubType := TypePtr;
    TypeTabl ^ [FwdPointer].TypeDesc := DeadFwdPtrType;
end;
end;
DecType(IdentNum,TypePtr);
Expect(semicolon1,14);
end;
end;

begin
ScanType := ScanTypeAct;
ScanConstDec := ScanConstDecAct;
ScanTypeDec := ScanTypeDecAct;
end.
unit sprfnvar;
{scans procedures, functions, and VAR declarations}

interface
uses
  getsym, {used to obtain next token}
  lex, {used to get status reports}
  labels, {used to generate internal labels and locations}
  lblock, {keeps track of block level}
  types, {declares all types used}
  options, {used to flag all options - call to check options}
  tables, {stores all compiler created tables}
  gencode, codeemit, {used to generate pseudo code}
  CircRef, {used for procedural variables}
  errors; {used to flag all errors}

{ procedure ScanProcDec;
  procedure ScanFuncDec;
  procedure ScanVarDec;
  procedure ScanDeclarations;
}

implementation

procedure ScanDeclarationsAct;
var ScanDone : boolean;
begin
  ScanDone := false;
  while (not ScanDone) do
    case LexSym.token of
      constl : ScanConstDec;
      typel : ScanTypeDec;
      procedurel : ScanProcDec;
      functionl : ScanFuncDec;
      varl : ScanVarDec;
      else ScanDone := true;
    end;
  end;
end;

procedure ScanProcDecAct;
var ProcLabel : longint; ParmCount: integer;
  IdDesc : IdRec; found:boolean; fwd:boolean;
  size : longint; IdPtrTmp : integer;
  row, col, fnum : integer;
  i : integer;
  IsException, IsRsrcProc : boolean;
  StartLabel, MemSize : longint;
  fpotr, TypSize : longint;
begin
  get; {clear out the PROCEDURE}
  fwd := false;
  IsRsrcProc := false;
  if LexSym.token <> identifierl then FlagError(2);
  GetStatus(row, col, fnum);
if AlreadyDef(LexSym.i1) then
begin
  GetIdDesc(LexSym.i1,IdDesc,found);
  if not (IdDesc.IdTyp in [DecRsrcProc,FwdProcName]) then
    FlagError(101)
  else begin
    IdPtrTmp := IdTablPtr;
    while IdTabl^[IdPtrTmp].IdNum <> LexSym.i1 do
dec(IdPtrTmp);
    case IdTabl^[IdPtrTmp].IdTyp of
      FwdProcName : IdTabl^[IdPtrTmp].IdTyp := ProcName;
      DecRsrcProc : begin
        IdTabl^[IdPtrTmp].IdTyp := UsedRsrcProc;
        IsRsrcProc := true;
      end;
    end;
    fptr := IdDesc.valptr;
    {re-declare the parameter names}
    StartNewBlock;
    if IsRsrcProc then
      for i := FieldTabl^[IdDesc.valptr].FTypePtr to
        FieldTabl^[IdDesc.valptr].FTypePtr +
        FieldTabl^[IdDesc.valptr].FTypePcnt-1 do
          DecParam(ParamTabl^[i].IdNum,i,ParamTabl^[i].IsVar)
    else
      for i := ProcTabl^[IdDesc.valptr].FirstParam to
        ProcTabl^[IdDesc.valptr].FirstParam + ProcTabl^[IdDesc.valptr].
        ParameterCount-1 do
        DecParam(ParamTabl^[i].IdNum,i,ParamTabl^[i].IsVar);
    fwd := true;
    if IsRsrcProc then
      ProcLabel := FieldTabl^[IdDesc.valptr].AddrOffs
    else ProcLabel := ProcTabl^[IdDesc.valptr].CodeLabel;
  end;
end;
if not fwd then
begin
  ProcLabel := NewLabel;
  StoreProcData(ProcLabel,ParamTablPtr+1);
  DecProc(LexSym.i1,ProcTablPtr);
  StartNewBlock;
end;
get;
if not fwd then begin
  if (LexSym.token=LeftPar1) then ScanParmDec(size,false)
else size := 0;
  ParmCount := ParamTablPtr + 1
  - ProcTabl^[ProcTablPtr].FirstParam;
  ProcTabl^[ProcTablPtr].ParameterCount := ParmCount;
  ProcTabl^[ProcTablPtr].ParamSize := size;
  expect(semicolon1,14);
end else begin
  if LexSym.token=LeftParl then FlagError(119);
  expect(semicolon1,14);
end;
if (LexSym.token=forwardl) then begin
  if fwd then FlagError(161);
  MarkForwardDec(row,col,fnum);
  get;
  expect(semicolon1,14);
end else begin
  IsException := false;
  if not IsRsrcProc and (LexSym.token = exceptionl) then begin
    if BlockLevel <> 1 then FlagError(284);
    if not RomCode then FlagError(285);
    get; {exception}
    IsException := true;
  end;
  ScanDeclarations;
  if (LexSym.token <> beginl) then FlagError(17);
  Emit(Initialize,0);
  Emit(declabel,ProcLabel);
  if IsException then Emit(exprocstart,VarAddrs[BlockLevel])
  else begin
    if IsRsrcProc then begin
      StartLabel := NewLabel;
      Emit(JmpAlways,StartLabel);
      { Skip the stack size - it's stored elsewhere.
        MemSize := DefaultStack + VarAddrs[BlockLevel];
        Emit(DecLong,MemSize);
      }
      {put these in instead, so the heading will be the same for
       processes and resources}
      Emit(DecLong,0);
      Emit(DecWord,FieldTabl ^ [fptr].FTypePcnt);
      TypSize := TypeTabl ^ [ParamTabl ^ [i].TypePointer].TypeSize;
      Emit(DecLong,TypSize);
      Emit(declabel,StartLabel);
      ForgetLabel(StartLabel);
      Emit(StartRsrcProc,VarAddrs[BlockLevel])
    end;
    Emit(declabel,ProcLabel);
  end;
end;
end else Emit(procstart,VarAddrs[BlockLevel]);
end;
Emit(InitPF,0);
NewExitLevel(NewLabel);
ScanCmpdStat;
Emit(DecLabel,ExitLabel);
ForgetLabel(ExitLabel);
EndExitLevel;
Emit(ClosePF,0);
expect(semicolon,14);
if IsException then Emit(exprocend,VarAddrs[BlockLevel])
else begin
  if IsRsrcProc then Emit(EndRsrcProc,VarAddrs[BlockLevel])
  else Emitfprocend,VarAddrs[BlockLevel]);
  end;
end;
EndBlock;
end;

procedure ScanFuncDecAct;
var FuncLabel : integer; ParmCount : integer; ftype:integer;
  IdDesc : IdRec; found:boolean; fwd:boolean;
  size : longint; IdPtrTmp : integer;
  row,col,fnum:integer;
  row2,col2,fnum2:integer;
begin
  get; {clear out the FUNCTION}
  fwd := false;
  if LexSym.token <> identifier then FlagError(2);
  GetStatus(row,col,fnum);
  if AlreadyDef(LexSym.il) then begin
    GetIdDesc(LexSym.il,IdDesc,found);
    if (IdDesc.IdTyp <> FwdFuncName) then begin
      FlagError(101);
      fwd := false;
    end else begin
      IdPtrTmp := IdTablPtr;
      while IdTabl^[IdPtrTmp].IdNum <> LexSym.il do dec(IdPtrTmp);
      IdTabl^[IdPtrTmp].IdTyp := FuncName;
      FuncLabel := FuncTabl^[IdDesc.valptr].CodeLabel;
      fwd := true;
    end;
  end;
  if not fwd then begin
    FuncLabel := NewLabel;
    StoreFuncData(FuncLabel,ParamTablPtr+1);
    DecFunc(LexSym.il,FuncTablPtr);
  end;
  StartNewBlock;
  get;
  if not fwd then begin

if (LexSym.token==LeftParl) then ScanParmDec(size,false)
else size := 0;
ParmCount := ParamTablPtr + 1
- FuncTabl^[ FuncTablPtr].FirstParam;
FuncTabl^[ FuncTablPtr].ParameterCount := ParmCount;
expect(colon1,5);
GetStatus(row2,col2,fnum2);
ScanType(ftype);
if TypeTabl^[ ftype].level = BlockLevel then
  FlagErrorPos(190,row2,col2,fnum2);
if TypeTabl^[ ftype].TypeDesc=FwdPtrType then
  FlagErrorPos(118,row2,col2,fnum2);
if not(TypeTabl^[ ftype].TypeDesc in [IntType,LongintType,
  ShortintType,ByteType,WordType,RealType,DoubleType,PtrType,
  CharType,BoolType,StringType,EnumType,SubrangeType]) then
  FlagErrorPos(120,row2,col2,fnum2);
FuncTabl^[ FuncTablPtr].TypePointer := ftype;
FuncTabl^[ FuncTablPtr].RsltOffset := size;
size := size + TypeTabl^[ ftype].TypeSize;
size := (size + 1) and $fffffffe;
FuncTabl^[ FuncTablPtr].ParamSize := size;
expect(semicolon1,14);
e else begin
  if LexSym.token==LeftParl then FlagError(119);
  if LexSym.token==colon1 then FlagError(122);
  expect(semicolon1,14);
end;
e if (LexSym.token==forwardl) then begin
  if fwd then FlagError(161);
  MarkForwardDec(row,col,fnum);
  get;
  expect(semicolon1,14);
e end else begin
  ScanDeclarations;
e if (LexSym.token <> begin1) then FlagError(17);
  Emit(Initialize,0);
  Emit(declabel,FuncLabel);
  Emit(funcstart,VarAddrs[BlockLevel]);
  Emit(InitPF,0);
  NewExitLevelNewLabel;
  ScanCmpdStat;
  Emit(DecLabel,ExitLabel);
  ForgetLabel(ExitLabel);
  EndExitLevel;
  Emit(ClosePF,0);
  expect(semicolon1,14);
  Emit(funcend,VarAddrs[BlockLevel]);
e end;
EndBlock;
e end;
procedure ScanVarDecAct;
const Maxlds = 50;
var count, check : integer; IdNums : array[1..MaxIds] of integer;
        PointerToType : integer; row, col, fnum : integer;
        IdDesc : IdRec; found : boolean;
begin
    {clear out the VAR}
    get;
    while (LexSym.token = identifier) do begin
        count := 0;
        if not AlreadyDef(LexSym.il) then begin
            inc(count);
            IdNums[count] := LexSym.il;
            get;
        end else FlagError(101);
    end;
    expect(colon, 5);
    GetStatus(row, col, fnum);
    ScanType(PointerToType);
    if TypeTabl^[PointerToType].TypeDesc = FwdPtrType then
        FlagErrorPos(118, row, col, fnum);
    if LexSym.token = absolute then begin
        if count <> 1 then FlagError(281);
        get; {absolute}
        StoreVarData(PointerToType, VarAddrs[BlockLevel]);
        DeclareVar(IdNums[1], VarTablPtr);
    case LexSym.token of
        LongIntConst : begin
            VarTabl^[VarTablPtr].AbsOffset := LexSym.il;
            VarTabl^[VarTablPtr].AbsLevel := -1;
            VarTabl^[VarTablPtr].IsAbsolute := AbsVar;
        end;
        identifier : begin
            GetIdDesc(LexSym.il, IdDesc, found);
            if not found then FlagError(104);
            case IdDesc.IdTyp of
                LongIntConst : begin
                    VarTabl^[VarTablPtr].AbsOffset := IdDesc.valptr;
                end;
        end;
VarTabl^[VarTablPtr].
AbsLevel := -1;
VarTabl^[VarTablPtr].
IsAbsolute := AbsVar;
end;
VarName :
if VarTabl^[IdDesc.valptr].IsAbsolute = NotAbs then begin
VarTabl^[VarTablPtr].AbsOffset :=
VarTabl^[IdDesc.valptr].AddrOffset;
VarTabl^[VarTablPtr].AbsLevel :=
VarTabl^[IdDesc.valptr].level;
VarTabl^[VarTablPtr].IsAbsolute :=
AbsVar;
end else begin
VarTabl^[VarTablPtr].AbsOffset :=
VarTabl^[IdDesc.valptr].AbsOffset;
VarTabl^[VarTablPtr].AbsLevel :=
VarTabl^[IdDesc.valptr].AbsLevel;
VarTabl^[VarTablPtr].IsAbsolute :=
VarTabl^[IdDesc.valptr].IsAbsolute;
end;
ValParamName : begin
VarTabl^[VarTablPtr].AbsOffset :=
ParamTabl^[IdDesc.valptr].AddrOffs;
VarTabl^[VarTablPtr].AbsLevel :=
IdDesc.level;
VarTabl^[VarTablPtr].IsAbsolute :=
AbsValParam;
end;
VarParamName : begin
VarTabl^[VarTablPtr].AbsOffset :=
ParamTabl^[IdDesc.valptr].AddrOffs;
VarTabl^[VarTablPtr].AbsLevel :=
IdDesc.level;
VarTabl^[VarTablPtr].IsAbsolute :=
AbsVarParam;
end;
else FlagError(282);
end;
end;
else FlagError(282);
end;
else FlagError(282);
end;
get; {move on to the ;}
begin
  ScanProcDec := ScanProcDecAct;
  ScanFuncDec := ScanFuncDecAct;
  ScanVarDec := ScanVarDecAct;
  ScanDeclarations := ScanDeclarationsAct;
end.
unit getsym;
{used to obtain next token}

interface
uses crt, lex, errors, types;
var
  LexSym: LexOutput;

procedure get;
procedure PushBack(var L: LexOutput);
procedure expect(sym: LexSymbol; ErrNum: integer);

implementation

var
  PushBackPtr: word;
  PushBackStack: array[1..10] of LexOutput;

procedure get;
beging
  if not error then begin
    if PushBackPtr <> 0 then begin
      LexSym := PushBackStack[PushBackPtr];
      dec(PushBackPtr);
    end else begin
      GetToken(LexSym);
      if LexSym.token = error1 then FlagError(LexSym.i1);
    end;
  end;
end;

procedure PushBack(var L: LexOutput);
begin
  inc(PushBackPtr);
end;

procedure expect(sym: LexSymbol; ErrNum: integer);
begin
if LexSym.token = sym then get
else FlagError(ErrNum);
end;

begin
  PushBackPtr := 0;
end.
unit lex;
{reads in files, converts to lex tokens}

interface
uses dos,crt,strstore,types,options,list,errors,tables,gencode,codeemit;

procedure AssignInputFile(AssignName:DataString);
function GetIdStr(Indx:longint):string;
procedure Enablekill;
procedure GetToken(var LexToken : LexOutput);
procedure GetStatus(var row,column:integer; var fnum:integer);
procedure InitLexTables;
procedure DiscardLexTables;

var
  TotalLines : longint;

implementation

CONST
  MaxNumSymbols = 1000; {max # of identifiers & rsrvd. words}
  HashTableSize = 8192; {max # of characters in all idents & rsrvd words}
  apost = #39; {ASCII code for apostrophe}
  CR = #13; {ASCII code for CARRIAGE RETURN}
  LF = #10; {ASCII code for LINE FEED}
  TAB = #9; {ASCII code for TAB}
  MaxInclude = 8; {maximum nesting level for include files}

TYPE
  SymbolDesc = record {info used to describe rsrvd words, identifiers}
    HashPtr : longint; {pointer into word table for this entry}
    WordLength : longint; {number of characters in this entry}
    HashVal : longint; {actual hash value for this entry}
    SymbolType : LexSymbol; {token describing type of word}
    CapsMask : longint; {bit field for caps, 1=Upcase}
    valid : boolean; {indicates if this entry is in use}
  end;

  InputDataType = record
    FileNum : integer; {number of the file, used for error reporting}
    InFileName : string[64]; {name of the file}
    LineNum : word; {line currently being read from}
    DataPtr : word; {points to next character to be read}
  end;
Var
SymbolTable : SymTblPtr;  {identifier list}
HashTable : HashTblPtr;  {spellings of identifiers}
HashTablePtr : longint;  {pointer to next position in HashTable}
SymTblCount : longint;  {# of symbols in SymbolTable}
input : char;  {character read in from source code}
InputFile : array[1..MaxInclude] of text;  {source code file(s)}
OutputToken : LexOutput;
FileEnd : boolean;  {set by input routine when EOF reached}
FileLevel : longint;  {1=source,2=include,3=nested include, etc.}
InputStrings : InputDataPtr;
TempRow,TempCol : array[0..2] of longint;  {saved row & col}
DotDot : boolean;
reg : registers;
alias1 : pointer;
alias2 : array[0..1] of word absolute alias1;
StatusLineNum,StatusColNum : word;
Kill : boolean;

Procedure Enablekill;
begin
  Kill := true;
end;

Function BitMask(BitNum:byte):longint;
var
  Tmp : longint;
begin
  if BitNum > 31 then BitMask := 0
  else begin
    Tmp := 1;
    while BitNum > 0 do begin
      dec(BitNum);
      Tmp := Tmp shl 1;
    end;
    BitMask := Tmp;
  end;
end;
function LowCase(c:char):char;
begin
  if c in ['A'..'Z'] then
    LowCase := chr(ord(c) + ord('a') - ord('A'))
  else LowCase := c;
end;

function GetIdStr(Indx:longint):string;
var Tmp : string;
i,j : longint;
begin
  Tmp := ",
  with SymbolTable ^ [Indx] do
  if Valid then begin
    for i := HashPtr to HashPtr + WordLength - 1 do
      Tmp := Tmp + HashTable ^ [i];
    for j := 1 to WordLength do
      if (CapsMask AND BitMask(j-1)) = 0 then Tmp[j] := LowCase(Tmp[j]);
  end;
  GetIdStr := Tmp;
end;

procedure InitHash; {Used to clear hash table, initialize pointer,counter}
var irlongint;
begin
  {mark each entry in SymbolTable as empty}
  for i := 0 to MaxNumSymbols-1 do SymbolTable ^ [i].valid := false;
  HashTablePtr := 0; {store first identifier in spelling table at location 0}
  SymTblCount := 0; {SymbolTable is empty}
end; {procedure InitHash}

function HashFnct(s:DataString):longint; {longint function of a string}
const weights : array[1..8] of word = (8,7,6,5,4,3,2,1);
var i,f,l:longint; {NOT case sensitive}
begin
  f := 0;
l := length(s);
  if l > 8 then l := 8;
  for i := 1 to l do
    f := f + weights[i]*ord(upcase(s[i]));
  for i := 9 to length(s) do
    f := f + ord(upcase(s[i]));
  HashFnct := f mod MaxNumSymbols; {use MODulo to keep it small}
end; {function HashFnct}

procedure InsertSymbol(s:DataString;t:LexSymbol);
{used to add reserved words to the symbol table}
var i,j:longint;
begin
  SymTblCount := SymTblCount + 1; {increment count}
  if SymTblCount >= MaxNumSymbols then begin {check for table overflow}

writeln('ERROR - Symbol Table Full');
halt(5);
end else begin
i := HashFnct(s); {try to insert word at SymbolTable[hash]}
{if that space is occupied, get first available space after it}
while SymbolTable^i.Valid do i := (i+1) mod MaxNumSymbols;
with SymbolTable^i do begin
valid := true; {mark entry as used}
HashTablePtr := HashTablePtr; {keep track of where char's are stored}
WordLength := length(s); {save word length}
SymbolType := t; {save type of reserved word}
HashVal := HashFnct(s); {save hash value for future searches}
CapsMask := 0; {calc the caps mask}
for i := 1 to length(s) do
if s[i] = UpCase(s[i]) then CapsMask := CapsMask OR BitMask(i-1);
end;
{check for spelling table overflow}
if (HashTablePtr + length(s)) >= HashTableSize then
writeln('ERROR - Hash Table Full');
else begin {otherwise, move string into spelling table}
for i := 1 to length(s) do
HashTable^HashTablePtr+i := UpCase(s[i]);
HashTablePtr := HashTablePtr + length(s);
end;
end;{procedure InsertSymbol}

procedure find(s:DataString;var t:LexSymbol;var idongint);
{used to search for a reserved word or an identifier in the symbol table}
function StrMatch:boolean; {local function - used to compare strings}
var j:longint; m:boolean;sl,s2:DataString;
begin
m := true; {assume a match}
with SymbolTable^i do {we will be checking entry i of the symbol table}
begin
if HashFnct(s) <> HashVal then m := false {no match if hash's differ}
else
if WordLength <> length(s) then m := false {nor if length's differ}
else
begin {hash & length match - extract string & compare}
s1 := ""; {clear string}
for j := HashPtr to (HashTablePtr+WordLength-1) do
s2 := s2 + HashTable^j; {extract char's & build up s2}
s1 := ""; {copy string of interest into s1, using upper case}
for j := 1 to WordLength do s1 := s1 + UpCase(s[j]);
if s1 <> s2 then m := false; {make final comparison}
end;
end;
StrMatch := m; {assign result of all comparisons}
begin {procedure find}
i := HashFnct(s); \{start looking at SymbolTable [HashFnct]\}
{step through the symbol table until a match is found or an empty
entry in the table is discovered}
while SymbolTable \[i\].valid and not StrMatch do
i := (i+1) mod MaxNumSymbols;
{if an empty entry stopped the search, then the word is not in the
symbol table, so add it using InsertSymbol}
if not SymbolTable \[i\].valid then InsertSymbol(s,IDENTIFIER1);
{whether found or just added, i points to the correct entry in the table}
t := SymbolTable \[i\].SymbolType; \{return SymbolType\}
{i is returned as a var parameter}
end; \{procedure find\}

procedure DoFileStart;
var i: integer;
begin
gotoxy(1,WhereY);
ClrEol;
for i := 1 to length(InputStrings \[FileLevel\].InFileName) do
write(Upcase(InputStrings \[FileLevel\].InFileName[i]));
write('0');
StatusLineNum := WhereY; StatusColNum := WhereX;
write(InputStrings \[FileLevel\].LineNum,')
end;

procedure ReadFromFile;
var DataFound : boolean;
begin
DataFound := false;
repeat \{Keep working way down until either the very end is reached or \}
{some characters are found to be read.\}
{first, see if the current level is at the end of file}
if eof(InputFile[FileLevel]) then
begin
close(InputFile[FileLevel]);
dec(FileLevel);
write('n');
if FileLevel < 1 then FileEnd := true
else DoFileStart;
end;
{next, make sure the current level is not exhausted }\nif not FileEnd then
if (InputStrings \[FileLevel\].DataPtr <=
InputStrings \[FileLevel\].DataSize) or
(not eof(InputFile[FileLevel])) then DataFound := true;
until DataFound or FileEnd;
if (not FileEnd) then
begin
if (InputStrings ^ [FileLevel].DataPtr >
  InputStrings ^ [FileLevel].DataSize) then
with InputStrings ^ [FileLevel] do
begin {read in more data into the appropriate file}
  DataPtr := 1;
  inc(LineNum);
  inc(TotalLines);
  gotoxy(StatusColNum,StatusLineNum);
  write(LineNum,' ');
  readln(InputFile[FileLevel],CharData);
  DataSize := length(CharData) + 1;
  CharData := CharData + CR;
end;
end;
end;

procedure GetChar;
var ListOutStr : string[128];
  i : word;
begin
  TempRow[2] := TempRow[1];
  TempCol[2] := TempCol[1];
  if InputStrings ^ [FileLevel].DataPtr >
    InputStrings ^ [FileLevel].DataSize then ReadFromFile;
  if not FileEnd then
  begin
    if InputStrings ^ [FileLevel].DataPtr=l then
      ListOutC + InputStrings[FileLevel].CharData);
    input := InputStrings [FileLevel].CharData[
      InputStrings ^ [FileLevel].DataPtr];
    inc(InputStrings ^ [FileLevel].DataPtr);
  end;
  if not FileEnd then
  begin
  end;
  if input = #26 then input := ' ';
end;

procedure EmitToken(t:LexSymbol);
begin
  OutputToken.token := t;
end;

procedure EmitLongint(i:longint);
begin
  OutputToken.token := LongintConst1;
  OutputToken.il := i;
end;
procedure EmitString(s:DataString);
begin
  if length(s) = 1 then
    begin
      OutputToken.token := CharConst1;
      OutputToken.i1 := ord(s[1]);
    end
  else
    begin
      OutputToken.token := StringConst1;
      SaveStringConstant(s,OutputToken.i1);
    end;
end;

procedure EmitReal(r: double);
begin
  OutputToken.token := RealConst1;
  SaveRealConstant(r,OutputToken.i1);
end;

procedure EmitError(i:longint);
begin
  OutputToken.token := ERROR1;
  OutputToken.i1 := i;
end;

procedure Scanldentifier;
var s:DataString;t:LexSymbol;i,extra:longint;
begin
  s := ";
  while (input in [a.'z','A.'Z',0.'9',_])
  and not FileEnd do
    begin
      s := s + input;
      GetChar;
    end;
  if Kill and (pos(' _',s) <> 0) then begin
    for i := 1 to length(s) do s[i] := LowCase(s[i]);
    while pos(' _',s) <> 0 do begin
      if pos(' _',s) <> length(s) then
        s[pos(' _',s) + 1] := UpCase(s[pos(' _',s) + 1]);
      delete(s,pos(' _',s),1);
      end;
    s[1] := UpCase(s[1]);
    end;
  find(s,t,i);
  OutputToken.token := t;
  if t=IDENTIFIER1 then OutputToken.i1 := i else OutputToken.i1 := extra;
end;

procedure ScanNumber;
var IntVal : longint;
  Exponent : longint;
  RealVal, Scale : double;
  IsInt, NegExp : boolean;
begin
  IsInt := true; {assume it is an integer}
  DotDot := false;
  IntVal := 0;
  RealVal := 0.0;
{calculate the integer portion of the value}
  while (input in ['0'..'9']) and not FileEnd do
  begin
    IntVal := 10 * IntVal + ord(input) - ord('0');
    RealVal := 10 * RealVal + ord(input) - ord('0');
    GetChar;
  end;
{see if it's a floating point value by checking for a decimal point}
  if input='.' then
  begin
    GetChar; {get rid of the decimal point}
    DotDot := input='.';
    if not DotDot then
    begin
      IsInt := false; {this is not an integer}
      Scale := 0.1; {used to scale the fractional portion}
      while (input in ['0'..'9']) and not FileEnd do
      begin
        RealVal := RealVal + Scale*(ord(input) - ord('0'));
        GetChar;
        Scale := Scale / 10.0;
      end;
    end;
  end;
{see if it's a floating point value by checking for an exponent}
  if ((input='e') or (input='E')) and (not DotDot) then
  begin
    IsInt := false; {this is not an integer}
    GetChar; {get rid of the e or E}
    {check for sign of exponent}
    case input of
      '-' : begin NegExp := true; GetChar; end;
      '+' : begin NegExp := false; GetChar; end;
      else NegExp := false;
    end;
{make sure a valid exponent follows}
  if not (input in ['0'..'9']) then
  begin
    FlagError(201);
    exit;
  end;
end;
{calculate the exponent}
Exponent := 0;
while (input in ['0'..'9']) and not FileEnd do
begin
   Exponent := 10 * Exponent + ord(input) - ord('0');
   GetChar;
   if abs(Exponent) > 500 then
   begin
      FlagError(207);
      exit;
   end;
end;
if NegExp then RealVal := RealVal / Exp(Exponent * Ln(10.0))
else RealVal := RealVal * Exp(Exponent * Ln(10.0));
end;
if Islnt then
   EmitLongint(IntVal)
else
   EmitReal(RealVal);
end;

procedure ScanOperator;
begin
   case input of
      '(' : begin
         EmitToken(LeftParl);
         GetChar;
      end;
      ')' : begin
         EmitToken(RightParl);
         GetChar;
      end;
      '-' : begin
         EmitToken(MINUSl);
         GetChar;
      end;
      '=' : begin
         EmitToken(EQUAL1);
         GetChar;
      end;
      '+' : begin
         EmitToken(PLUS1);
         GetChar;
      end;
      '*' : begin
         EmitToken(TIMESl);
         GetChar;
      end;
      '^' : begin
         EmitToken(carrot1);
         GetChar;
      end;
   end;
end;
end;
'[' : begin
  EmitToken(LeftBrkt1);
  GetChar;
end;
']' : begin
  EmitToken(RightBrkt1);
  GetChar;
end;
';' : begin
  EmitToken(SEMICOLON1);
  GetChar;
end;
':' : begin
  EmitToken(COLON1);
  GetChar;
end;
'/' : begin
  EmitToken(DIVIDE1);
  GetChar;
end;
'@' : begin
  EmitToken(ampersand1);
  GetChar;
end;
':' : begin
  GetChar;
  if input='=' then begin
    EmitToken(ASSIGN1);
    GetChar;
  end else EmitToken(COLON1);
end;
'<' : begin
  GetChar;
  if input='=' then begin
    EmitToken(LessEq1);
    GetChar;
  end else if input='>' then begin
    EmitToken(NotEqual1);
    GetChar;
  end else EmitToken(LessThen1);
end;
'>' : begin
  GetChar;
  if input='=' then begin
    EmitToken(GreaterEq1);
    GetChar;
  end else EmitToken(GreaterThen1);
end;
'.' : begin
  GetChar;
if DotDot then begin
    EmitToken(DotDot1);
    DotDot := false;
end else begin
    if input='.' then begin
        EmitToken(DotDot1);
        GetChar;
        end else EmitToken(period1);
    end;
end;
end;

procedure ScanString;
var s:DataString;error:longint;done:boolean;
begin
    s := ";
    error := 0;
    done := false;
    while not done do
    begin
        GetChar;
        if FileEnd then
        begin
            done := true;
            error := 1;
        end
        else
        case input of
        apost : begin
            GetChar;
            if input=apost then s := s + input
            else done := true;
        end;
        CR : begin
            done := true;
            error := 2;
        end;
        else s := s + input;
        end;
        if error <> 0 then FlagError(error);
        EmitString(s);
    end;

function FileExists(s:string):boolean;
var f:file of byte;
    IOR : longint;
begin
    assign(F,s);
    if IOResult <> 0 then ;
4 5 0

$I-$reset(F);$I+$

IOR := IOResult;
if IOR=0 then begin
  FileExists := true;
  close(F);
end else begin
  FileExists := false;
end;
end;

procedure IncludeFile;
var FileName:DataString;
begin
  FileName := "$;\{initialize to empty string\}"
  GetChar; \{get first character of file name\}
  if (input='+' ) or (input='.' ) then
    begin
      if input='.' then DisableOption('I')
      else EnableOption('I');
    end else begin
      while (input=' ' ) do GetChar; \{eliminate any leading spaces\}
      while (input <> '}' ) do begin
        FileName := FileName + input; \{build up file name\}
        GetChar; \{get next character\}
        if (input=CR) then input := '>'; \{check for end of line\}
      end;
      if not FileExists(FileName) then begin
        if FileExists(FileName+'.bpl') then FileName := FileName + '.BPL'
        else begin
          if FileExists(FileName+'.pas') then FileName := FileName + '.PAS'
          else begin
            TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
            FlagError(400);
            end;
          end;
        end;
      end;
      inc(FileLevel); \{indicate nested file\}
      InputStrings^ [FileLevel].DataPtr := 2; \{force read on next call\}
      InputStrings^ [FileLevel].DataSize := 1; \{ to GetChar\}
      InputStrings^ [FileLevel].LineNum := 0;
      InputStrings^ [FileLevel].InFileName := FileName;
      TempRow[1] := 0;
      TempCol[1] := 1;
      TempRow[2] := TempRow[1];
      TempCol[2] := TempCol[1];
      StoreFileName(FileName,InputStrings^ [FileLevel].FileNum);
      assign(InputFile^ [FileLevel].FileName);
      reset(InputFile^ [FileLevel]);
      DoFileStart;
      end;
end;
procedure StackOption;
var StackVal : longint;
begin
repeat
  GetChar;
until not (input in [' ','CR,LF,TAB']);
if not (input in ['0'..'9','$']) then FlagError(215);
StackVal := 0;
if input='='$' then
begin
  GetChar;
  repeat
    case UpCase(input) of
      'A'..'Z' : StackVal := 16 * StackVal + ord(Upcase(input)) - ord('A') + 10;
      '0'..'9' : StackVal := 16 * StackVal + ord(Upcase(input)) - ord('0');
      else FlagError(215);
    end;
  GetChar;
  until not (UpCase(input) in ['A'..'Z','0'..'9']);
end else repeat
StackVal := 10 * StackVal + ord(input) - ord('0');
GetChar;
until not (input in ['0'..'9']);
StoreStackSize(StackVal);
while input <> '}' do GetChar;
end;

procedure MachineOption;
var VarBase,CodeBase : longint;
begin
GetChar; {M}
while input in [' ','CR,LF,TAB] do GetChar;
TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
if not (input in ['0'..'9','$']) then FlagError(280);
VarBase := 0;
if input='='$' then
begin
  GetChar;
  repeat
    case UpCase(input) of
      'A'..'Z' : VarBase := 16 * VarBase + ord(Upcase(input)) - ord('A') + 10;
      '0'..'9' : VarBase := 16 * VarBase + ord(Upcase(input)) - ord('0');
      else FlagError(280);
    end;
  GetChar;
  until not (UpCase(input) in ['A'..'Z','0'..'9']);
end else repeat
VarBase := 10 * VarBase + ord(input) - ord('0');
GetChar;
until not (input in ['0'..'9']);
while input in ['
','\n','\r','\t'] do GetChar;
TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
if not (input in [',']) then FlagError(20);
GetChar;
while input in ['
','\n','\r','\t'] do GetChar;
TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
if not (input in ['0'..'9','$']) then FlagError(280);
CodeBase := 0;
if input='$' then begin
  GetChar;
  TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
  repeat
    case UpCase(input) of
      'A'..'Z' : CodeBase := 16 * CodeBase + ord(Upcase(input)) - 
                  ord('A') + 10;
      '0'..'9' : CodeBase := 16 * CodeBase + ord(Upcase(input)) - 
                  ord('0');
      else FlagError(280);
    end;
  GetChar;
  until not (UpCase(input) in ['A'..'Z','0'..'9']);
end end else repeat
  CodeBase := 10 * CodeBase + ord(input) - ord('0');
  GetChar;
  until not (input in ['0'..'9']);
if CheckOption('lC) then inc (VarBase,$2000);
SelectRomCode (VarBase, CodeBase);
while input <> '>' do GetChar;
end;

procedure NumRequestsOption;
var StackVal: longint;
begin
  repeat
    GetChar;
    until not (input in ['
','\n','\r','\t']);
  if input in ['-','+'] then begin
    case input of
      '-' : DisableOption('T');
      '+' : EnableOption('T');
    end;
    GetChar;
    exit;
  end;
  if not (input in ['0'..'9','$']) then FlagError(215);
  StackVal := 0;
  if input='$' then begin
    StackVal := StackVal + 1;
    while input <> '>' do GetChar;
  end;
end;
GetChar;
repeat
  case UpCase(input) of
    'A'..'Z' : StackVal := 16 * StackVal + ord(Upcase(input)) - ord('A') + 10;
    '0'..'9'  : StackVal := 16 * StackVal + ord(Upcase(input)) - ord('0');
  else FlagError(215);
  end;
  GetChar;
until not (UpCase(input) in ['A'..'Z','0'..'9']);
end else repeat
  StackVal := 10 * StackVal + ord(input) - ord('0');
  GetChar;
until not (input in ['0'..'9']);
NumRequests := StackVal;
while input <> '}' do GetChar;
end;

procedure ScanComment;
var level:longint;c:char;
begin
  GetChar;
  if (input = '$') then  {check for options}
  begin
    GetChar;           {see what option it is}
    input := UpCase(input);
    if input in ['A'..'Z'] then
      case input of
        'I'  : IncludeFile;
        'S' : StackOption;
        'M' : MachineOption;
        'T' : NumRequestsOption;
      else
        begin
          c := input;
          GetChar;
          if (input = '+') then EnableOption(c);
          if (input = '-') then DisableOption(c);
        end;
      end;
    level := 1;
  end;
  while (level>0) and (not FileEnd) do
  case input of
    ')' : begin
      level := level - 1;
      GetChar;
    end;
    '{' : begin
      level := level + 1;
      GetChar;
    end;
  end;
end;
end;
else GetChar;
end;
if FileEnd and not (level=0) then FlagError(3);
end;

procedure ScanHexChar;
var i:longint;
begin
  GetChar;
i := 0;
if not(input in ['0'..'9','a'..'f','A'..'F']) then FlagError(15)
else
  while (input in ['0'..'9','a'..'f','A'..'F']) and not FileEnd do begin
    case input of
      '0'..'9': i := 16*i + ord(input) - ord('0');
      'a'..'f': i := 16*i + ord(input) - ord('a') + 10;
      'A'..'F': i := 16*i + ord(input) - ord('A') + 10;
    end;
    GetChar;
  end;
OutputToken.token := CharConst1;
OutputToken.i1 := i;
end;

procedure ScanHexLongint;
var i:longint;
begin
  GetChar;
i := 0;
if not(input in ['0'..'9','a'..'f','A'..'F']) then FlagError(15)
else
  while (input in ['0'..'9','a'..'f','A'..'F']) and not FileEnd do begin
    case input of
      '0'..'9': i := 16*i + ord(input) - ord('0');
      'a'..'f': i := 16*i + ord(input) - ord('a') + 10;
      'A'..'F': i := 16*i + ord(input) - ord('A') + 10;
    end;
    GetChar;
  end;
OutputToken.token := LongintConst1;
OutputToken.i1 := i;
end;

procedure ScanChar;
var i:longint;
begin
  GetChar;
if input=' $' then ScanHexChar
else if not (input in ['0'..'9']) then FlagError(15)
else begin
i := 0;
while (input in ['0'..'9']) and not FileEnd do begin
  i := 10 * i + ord(input) - ord('0');
  GetChar;
end;
OutputToken.token := CharConst1;
OutputToken.i1 := i;
end;
end;

procedure ScanError;
begin
  FlagError(6);
  GetChar;
end;

procedure InitSymbols;
begin
  InsertSymbol('with',with1);
  InsertSymbol('begin',begin1);
  InsertSymbol('end',end1);
  InsertSymbol('and',and1);
  InsertSymbol('if',if1);
  InsertSymbol('WHILE',WHILE1);
  InsertSymbol('DO',DO1);
  InsertSymbol('PROGRAM',PROGRAM1);
  InsertSymbol('procedure',procedure1);
  InsertSymbol('function',function1);
  InsertSymbol('THEN',THEN1);
  InsertSymbol('for',for1);
  InsertSymbol('to',to1);
  InsertSymbol('downto',downto1);
  InsertSymbol('ELSE',ELSE1);
  InsertSymbol('WHILE',WHILE1);
  InsertSymbol('DO',DO1);
  InsertSymbol('REPEAT',REPEAT1);
  InsertSymbol('UNTIL',UNTIL1);
  InsertSymbol('case',case1);
  InsertSymbol('of',of1);
  InsertSymbol('var',var1);
  InsertSymbol('ret',ret1);
  InsertSymbol('type',type1);
  InsertSymbol('const',const1);
  InsertSymbol('integer',integer1);
  InsertSymbol('shortint',shortint1);
  InsertSymbol('longint',longint1);
  InsertSymbol('byte',byte1);
  InsertSymbol('word',word1);
  InsertSymbol('real',real1);
  InsertSymbol('double',double1);
  InsertSymbol('boolean',boolean1);
InsertSymbol('char',char1);
InsertSymbol('not',not1);
InsertSymbol('or',or1);
InsertSymbol('xor',xor1);
InsertSymbol('string',string1);
InsertSymbol('array',array1);
InsertSymbol('file',file1);
InsertSymbol('text',text1);
InsertSymbol('set',set1);
InsertSymbol('in',in1);
InsertSymbol('record',record1);
InsertSymbol('div',div1);
InsertSymbol('mod',mod1);
InsertSymbol('true',true1);
InsertSymbol('false',false1);
InsertSymbol('forward',forward1);
InsertSymbol('read',read1);
InsertSymbol('readln',readln1);
InsertSymbol('write',write1);
InsertSymbol('writeln',writeln1);
InsertSymbol('eof',eof1);
InsertSymbol('eoc',eoc1);
InsertSymbol('inc',inc1);
InsertSymbol('dec',dec1);
InsertSymbol('ord',ord1);
InsertSymbol('chr',chr1);
InsertSymbol('assign',assign1);
InsertSymbol('reset',reset1);
InsertSymbol('rewrite',rewrite1);
InsertSymbol('close',close1);
InsertSymbol('seek',seek1);
InsertSymbol('pi',pi1);
InsertSymbol('absolute',absolute1);
InsertSymbol('exception',exception1);
InsertSymbol('setexceptionvec',setexceptionvec1);
InsertSymbol('inline',inline1);
InsertSymbol('process',process1);
InsertSymbol('resource',resource1);
InsertSymbol('parallel',parallel1);
InsertSymbol('select',select1);
InsertSymbol('channel',channel1);
InsertSymbol('interface',interface1);
InsertSymbol('implementation',implementation1);
InsertSymbol('sqrt',sqrt1);
InsertSymbol('sqf',sqrl);
InsertSymbol('sin',sin1);
InsertSymbol('round',round1);
InsertSymbol('cos',cos1);
InsertSymbol('tan',tan1);
InsertSymbol('arccos',acos1);
InsertSymbol('arcasin',asin1);
InsertSymbol('arccos',acos1);
procedure AssignInputFile(AssignName:DataString);
begin
  FileLevel := 1; {source file}
  if not FileExists(AssignName) then begin
    if FileExists(AssignName + '.bpF') then
      AssignName := AssignName + '.BPL'
    else begin
      if FileExists(AssignName + '.PAS') then
        AssignName := AssignName + '.PAS'
      else begin
        writeln('ERROR: File ',AssignName,' Not Found.');
        halt(5);
      end;
    end;
  end;
  assign (InputFile[FileLevel].AssignName);
  reset(InputFile[FileLevel]);
  StoreFileName(AssignName,InputStrings^ [FileLevel].FileNum);
  FileEnd := false;
  InputStrings^ [FileLevel].LineNum := 0;
  InputStrings^ [FileLevel].DataPtr := 2;
  InputStrings^ [FileLevel].DataSize := 1;
  InputStrings^ [FileLevel].InFileName := AssignName;
  TempRow[1] := 0;
  TempCol[1] := 1;
  InitHash;
  InitSymbols;
  DoFileStart;
  GetChar;
end;

procedure ScanLex;
begin
if not FileEnd then begin
  TempRow[0] := TempRow[2]; TempCol[0] := TempCol[2];
  case input of
    'A..'Z', 'a..'z': ScanIdentifier;
    '0..'9': ScanNumber;
    '(', ')', '{', '}', '!', '+', '-' : ScanOperator;
    ':=' : ScanAssign;
    ':' : ScanComment;
    '(': begin
      ScanComment;
      ScanLex;
    end;
    '#': ScanChar;
    '$': ScanHexLongint;
    CR, LF, TAB : begin
      while not FileEnd AND (Input in [CR, LF, TAB]) do GetChar;
      ScanLex;
    end;
    else ScanError;
  end;
end else FlagError(64);
end;

procedure GetToken(var LexToken : LexOutput);
begin
  ScanLex;
  LexToken := OutputToken;
end;

procedure GetStatus(var row,column:integer;
  var fnumber:integer);
begin
  row := TempRow[0];
  column := TempCol[0];
  if FileLevel=0 then fnumber := InputStrings^1.FileNum
  else fnumber := InputStrings^[FileLevel].FileNum;
end;

procedure InitLexTables;
begin
  new(SymbolTable);
  new(HashTable);
  new(InputStrings);
  DotDot := false;
  Kill := false;
  TotalLines := 0;
end;

procedure DiscardLexTables;
begin
  dispose(SymbolTable);
  dispose(HashTable);
  dispose(InputStrings);
  while FileLevel > 0 do begin
    close(InputFile[FileLevel]);
    dec(FileLevel);
  end;
end;

unit CodeEmit;

interface

uses types, tables, errors, list, strstore, options, labels, gencode, crt,
  CircRef,         {used for procedural variables}
  EmitInit,
  EmitCnt,
  EmitFor,
  EmitMath,
  EmitStr,
  EmitLog,
  EmitFile,
  EmitPar,
  EmitMem,
  EmitShr;

procedure EmitAct(instr: OpCode; param: longint);

implementation

procedure EmitAct(instr: OpCode; param: longint);
var LablName: string[25];
  i: integer;
  RConst: double;
  RSize: longint;
  CVal: longint;
  CStr: string[25];
  X, Y: byte;
  c: char;
begin
  if CheckOption('D') then begin
    x := WhereX;
    y := WhereY;
    window(25, 6, 80, 20);
    clrscr;
    write('Instruction: ');
    case instr of
      Initialize: writeln('Initialize');
      DecLabel: writeln('DecLabel');;
BitwiseNot : writeln('BitwiseNot');
CmpEQ : writeln('CmpEQ');
CmpNE : writeln('CmpNE');
CmpLT : writeln('CmpLT');
CmpLE : writeln('CmpLE');
CmpGT : writeln('CmpGT');
CmpGE : writeln('CmpGE');
StrConstant : writeln('StrConstant');
StoreStr : writeln('StoreStr');
StoreCharInStr : writeln('StoreCharInStr');
PushStrValParam : writeln('PushStrValParam');
FetchStr : writeln('FetchStr');
PushCharAsStrParam : writeln('PushCharAsStrParam');
AddStrings : writeln('AddStrings');
AddCharStr : writeln('AddCharStr');
AddStrChar : writeln('AddStrChar');
AddCharChar : writeln('AddCharChar');
CharToStr : writeln('CharToStr');
ExpandFuncStr : writeln('ExpandFuncStr');
InitStr : writeln('InitStr');
CloseStr : writeln('CloseStr');
AssignFile : writeln('AssignFile');
ResetFile : writeln('ResetFile');
RewriteFile : writeln('RewriteFile');
CloseFile : writeln('CloseFile');
SeekFile : writeln('SeekFile');
SelectStdIO : writeln('SelectStdIO');
SelectIO : writeln('SelectIO');
ReleaseIO : writeln('ReleaseIO');
ChkEOF : writeln('ChkEOF');
ChkEOC : writeln('ChkEOC');
PrintInt : writeln('PrintInt');
PrintReal : writeln('PrintReal');
PrintChar : writeln('PrintChar');
PrintString : writeln('PrintString');
PrintEol : writeln('PrintEol');
Address2Int : writeln('Address2Int');
ReadFile : writeln('ReadFile');
WriteFile : writeln('WriteFile');
ClrInputLine : writeln('ClrInputLine');
ReadVar : writeln('ReadVar');
ExProcStart : writeln('ExProcStart');
ExProcEnd : writeln('ExProcEnd');
GetLabelAddress : writeln('GetLabelAddress');
StoreVector : writeln('StoreVector');
DecWord : writeln('DecWord');
DecLong : writeln('DecLong');
StartProcess : writeln('StartProcess');
EndProcess : writeln('EndProcess');
StartRsrcProc : writeln('StartRsrcProc');
EndRsrcProc : writeln('EndRsrcProc');
ParallelCall: writeln('ParallelCall');
CallRsrcProc: writeln('CallRsrcProc');
DoSin: writeln('DoSin');
DoCos: writeln('DoCos');
DoTan: writeln('DoTan');
DoASin: writeln('DoASin');
DoACos: writeln('DoACos');
DoATan: writeln('DoATan');
DoExp: writeln('DoExp');
DoLn: writeln('DoLn');
DoSqrt: writeln('DoSqrt');
DoSqr: writeln('DoSqr');
DoRound: writeln('DoRound');
DoAbs: writeln('DoAbs');
CmpStrs: writeln('CmpStrs');
CmpStrChar: writeln('CmpStrChar');
CmpCharStr: writeln('CmpCharStr');
DoMove: writeln('DoMove');
DoFillChar: writeln('DoFillChar');
ForPrelim: writeln('ForPrelim');
EndOfAnd: writeln('EndOfAnd');
EndOfOr: writeln('EndOfOr');
InitPF: writeln('InitPF');
ClosePF: writeln('ClosePF');
PrepareTemplateWrite: writeln('PrepareTemplateWrite');
PrepareTemplateRead: writeln('PrepareTemplateRead');
ReqRead: writeln('ReqRead');
ReqWrite: writeln('ReqWrite');
TempRelease: writeln('TempRelease');
FinishTemplateCheck: writeln('FinishTemplateCheck');
TempRangeStore: writeln('TempRangeStore');
FinishTemplateOp: writeln('FinishTemplateOp');
ChkTempIndex: writeln('ChkTempIndex');
ChkTempIndexRow: writeln('ChkTempIndexRow');
FlushIO: writeln('FlushIO');
end;
writeln('Before: ');
writeln('Virtual Stack Pointer: ', SP);
for i := SP downto 0 do begin
  write('Entry: ', i);
case CStack[i].SType of
  Empty: write('Empty':16);
  ConstOnly: write('ConstOnly':16);
  RegOnly: write('RegOnly':16);
  ConstPlusReg: write('ConstPlusReg':16);
  ConstTimesReg: write('ConstTimesReg':16);
  StrOnly: write('StrOnly':16);
  CondC: write('CondC':16);
  NewDead: write('NewDead':16);
  OldDead: write('OldDead':16);
  else write('ERROR':16);
end;
if CStack[i].IsReal then write(CStack[i].ConstVal:10:4)
  else write(GBTrunc(CStack[i].ConstVal):10);
case CStack[i].BaseAddr of
  NoBase : write('NoBase':11);
  VarAddr : write('VarAddr':11);
  ParamAddr : write('ParamAddr':11);
end;
writeln(CStack[i].BLevel);
end;

Simplify;
case instr of
  DecLabel,
  JmpTrue,
  JmpFalse,
  JmpAlways,
  Trace,
  ProgStart,
  ProgEnd,
  SourceCodeEnd,
  ProcStart,
  ExProcStart,
  ProcEnd,
  ExProcEnd,
  FuncStart,
  FuncEnd,
  ProcCall,
  FuncCall,
  PrepFuncCall : EmitControl(instr,param);

  EvalCase,
  CaseCmp,
  ForInc,
  ForDec,
  ToCmp,
  DowntoCmp,
  ForEnd,
  EnablePar,
  DisablePar,
  ForPrelim : EmitForCase(instr,param);

  NegateVal : negate;

  GetLabelAddress,
  FixStack,
  Address2Int,
  StoreVector,
  DecWord,
  DecLong,
  CopyParms,
PushBaseVarAddr,
PushBaseParamAddr,
PushStruct,
CopyStruct,
PushVal,
PushVar,
DoMove,
DoFillChar,
Fetch,
Store : EmitMemInstr(instr,param);

IncVar,
DecVar,
Add,
Subtract,
Mult,
Divide,
IntDiv,
Modulo,
IntConstant,
BoolConstant,
RealConstant,
AddConstant,
MultConstant,
DoSin,
DoCos,
DoTan,
DoACos,
DoASin,
DoATan,
DoExp,
DoLn,
DoSqrt,
DoSqr,
DoRound,
DoAbs,
ChkRange : EmitMathInstr(instr,param);

ChkStrIndex,
CmpStrs,
CmpStrChar,
CmpCharStr,
StrConstant,
StoreStr,
PushStrValParam,
PushCharAsStrParam,
FetchStr,
StoreCharInStr,
AddStrings,
AddCharStr,
AddStrChar,
AddCharChar,
CharToStr,
ExpandFuncStr,
InitStr,
CloseStr : EmitStrInstr(instr,param);

ParallelCall,
CallRsrcProc,
StartProcess,
EndProcess,
StartRsrcProc,
EndRsrcProc : EmitParInstr(instr,param);

LogAnd,
LogOr,
LogXor,
LogNot,
BitwiseNot,
EndOfAnd,
EndOfOr,
CmpEQ,
CmpNE,
CmpLT,
CmpLE,
CmpGT,
CmpGE : EmitLogCmp(instr,param);

AssignFile,
ResetFile,
RewriteFile,
CloseFile,
PrintInt,
PrintReal,
PrintChar,
PrintString,
PrintEol,
ReadFile,
ReadVar,
WriteFile,
ClrInputLine,
SelectStdIO,
SelectIO,
ReleaseIO,
SeekFile,
FlushIO,
ChkSelect,
SkipElse,
ChooseSelect,
PushLabelAddr,
ChkEOC,
ChkEOF : EmitFileInstr(instr,param);
\text{InitPF,}
\text{ClosePF : EmitInitialization(instr,param);

PrepareTemplateWrite, PrepareTemplateRead, ReqRead, ReqWrite, FinishTemplateCheck, FinishTemplateOp, TempRangeStore, ChkTempIndex, ChkTempIndexRow, TempRelease : EmitSharedVar(instr,param);

Initialize : SP := -1;
else FlagError(-31);
end;
if CheckOptionCD' then begin
\text{writeln('After : ');
\text{writeln('Virtual Stack Pointer : ',SP);
for i := SP downto 0 do begin
\text{write('Entry: ',i);
\text{case CStack[i].SType of
\text{Empty : write('Empty':16);
\text{ConstOnly : write('ConstOnly':16);
\text{RegOnly : write('RegOnly':16);
\text{ConstPlusReg : write('ConstPlusReg':16);
\text{ConstTimesReg : write('ConstTimesReg':16);
\text{StrOnly : write('StrOnly':16);
\text{CondC : write('CondC':16);
\text{NewDead : write('NewDead':16);
\text{OldDead : write('OldDead':16);
else write('ERROR':16);
end;
if CStack[i].IsReal then write(CStack[i].ConstVal:10:4)
else write(GBTrunc(CStack[i].ConstVal:10:4);
case CStack[i].BaseAddr of
\text{NoBase : write('NoBase':11);
\text{VarAddr : write('VarAddr':11);
\text{ParamAddr : write('ParamAddr':11);
end;
\text{writeln(CStack[i].BLevel);
end;
c := ReadKey;
window(1,1,80,25);
gotoxy(X,Y);
end;}
begin
Emit := EmitAct;
end.
unit emitshr;
interface
uses genode,types,tables,errors,list,strstore,options,labels,CircRef;
procedure EmitSharedVar(instr:OpCode;Param:longint);

implementation

procedure EmitSharedVar(instr:OpCode;Param:longint);
var L1,L2,CVal,CVal2 :  longint;
    CStr,CStr2 :  string[8];
begin
  case instr of
    PrepareTemplateWrite,
    PrepareTemplateRead : begin
      MultlnConstant(SP);
      GetFullAddr(SP);
      PushEA;
      {The address of the template is now on the cpu stack. First, check the word at that address
to see if the status is correct.}
      CMove(L,'(a7)', 'a0');
      CMove(W,'(a0)', 'd' + TopStr);
      if instr=PrepareTemplateWrite then
        CListOut('andi.l #2,d'+TopStr,3,
          $0280+nTop,0,2,0)
      else CListOut('andi.l #1,d'+TopStr,3,
          $0280+nTop,0,1,0);
{if the zero flag is set, we have an error}
      LI := NewLabel;
      DoBranch(CondNE,Ll);
      DoMoveQ(l,0); {error number}
      OSFunc(l); {run-time error function}
      PlaceForgetLabel (LI);
      { a0 now points to the template address.
      Read a long word from the template address+$0c.
      This is where the address of the data is stored. }
      decSP; incSP; {make space to store ptr to data}
      CMove(L,'12(a0)', 'd' + TopStr);
      CStack[SP].SType := RegOnly;
      { The base address of the shared variable is
      now on top of the virtual operand stack. The address
      of the template remains on the cpu stack. }
  end;
  ChkTempIndexRow : begin
    if CStack[SPm1].SType <> ConstOnly then
      FlagError(-530);
    if CStack[SP].SType <> ConstOnly then
      \text{...}
FlagError(-530);
L1 := NewLabel;
L2 := NewLabel;
{ Param is the index # of the range to be checked. }
CVal2 := $10 + (Param-1) * 8; str(CVal2,CStr2);
CVal := GBTTrunc(CStack[SPm1].ConstVal);
str(Cval,CStr);
CCmp(L,'# '+ CStr,CStr2+ ' (aO) ');
DoBranch(CondNE,L1);
inc(CVal2,4); str(CVal2,CStr2);
CVal := GBTTrunc(CStack[SP].ConstVal); str(Cval,CStr);
CCmp(L,'# '+ CStr,CStr2+ ' (aO) ');
DoBranch(CondEQ,L2);
PlaceForgetLabel(L1);
DoMoveQ(1,1); {error number}
OSFunc(1); {run-time error function}
PlaceForgetLabel(L2);
decSP; decSP;
end;
FinishTemplateCheck : CAdd(L,'#47a7' );
ChkTempIndex : begin
{Param= index number (1..NumArrays) }
ExpandCC(SP);
MultInConstant(SP);
AddBaseAddr(SP);
AddInConstant(SP);
CMove(L,'(a7),a0' );
CVal := $10 + (Param-1) * 8; str(CVal,CStr);
CListOut('chk2.1 '+CStr+'(a0),d'+TopStr,3,$04d0,
$0800+$1000*nTop,LoWord(CVal),0);
end;
ReqRead,
ReqWrite : begin
{Push the address of the template}
MultInConstant(SP);
GetFullAddr(SP);
PushEA;
decSP;
{Push the address of the shared variable}
MultInConstant(SP);
GetFullAddr(SP);
PushEA;
decSP;
{ get the template address into aO }
CMove(L,'4(a7),a0' );
{First, make sure the template is released.}
CListOut('tst.w (a0),1,$4a50,0,0,0);
L1 := NewLabel;
DoBranch(CondEQ,L1);
CMove(W,'#0', (a0)');
{number of dimensions into d1}
DoMoveQ(1,TypeTabl^[Param].P1);
OSFunc($52);
PlaceForgetLabel(L1);

{store the appropriate command word}
if instr = ReqWrite then CMove(W,'#3', (a0)')
else CMove(W,'#1', (a0)');

{Move the shared var ID into the template}
CMove(L,'(a7) +', '8(a0)');

{ The template address remains on the cpu stack. }
end;
FinishTemplateOp : begin
{ Remove template address from stack, store in a0. }
CMove(L,'(a7) +', 'a0');
DoMoveQ(1,TypeTabl^[Param].P1); { # dimensions into d1}
OSFunc($52);
end;
TempRangeStore : begin
{Param is # of range pair (1..NumArrays) }
{two range values are on the stack}
ExpandCC(SP);
MultInConstant(SP);
AddBaseAddr(SP);
ClearAddIns(SP);
ExpandCC(SPml);
MultInConstant(SPml);
AddBaseAddr(SPml);
ClearAddIns(SPml);
CMove(L,'(a7)', 'a0');
CVal := $10 + 8 * (Param-1); str(CVal,CStr);
CAdd(L,'#' + CStr, (a0)');
if CStack[SPml].SType=ConstOnly then begin
  CVal := GBTrunc(CStack[SPml].ConstVal);
  str(CVal,CStr);
  CMove(L,'#' + CStr, (a0)');
end else CMove(L,'d'+NextStr,'(a0)');
if CStack[SP].SType=ConstOnly then begin
  CVal := GBTrunc(CStack[SP].ConstVal);
  str(CVal,CStr);
  CMove(L,'#' + CStr, (a0)');
end else CMove(L,'d'+TopStr,'(a0)');
decSP; decSP;
end;
TempRelease : begin
  MultInConstant(SP);
  GetFullAddr(SP);
  CMove(L, AddrStr, 'a0');
  CListOut('test.w (a0)',1,$4a50,0,0,0);
  L1 := NewLabel;
  DoBranch(CondEQ,L1);
  CMove(W,'#0', (a0)));
{number of dimensions}
  DoMoveQ(1,TypeTabl ^ [Param].P1);
  OSFunc($52);
  PlaceForgetLabel(L1);
  decSP;
  end;
  else FlagError(-32);
  end;
end;

unit EmitMem;

interface
uses gencode, types, tables, errors,
  list, strstore, options, labels,
  CircRef; {used for procedural variables}

procedure EmitMemInstr(instr: OpCode; Param: longint);

implementation

procedure EmitMemInstr(instr: OpCode; Param: longint);
var LablName : string[25];
  i : integer;
  RConst : double;
  RSize : longint;
  CVal : longint;
  CStr : string[25];
  L1, L2 : longint;
begin
  case instr of
    PushBaseVarAddr: begin
      IncSP;
      with CStack[SP] do
        if RomCode and (Param=0) then begin
          SType := ConstOnly;
          ConstVal := RomVarBase;
        end else begin
          BaseAddr := VarAddr;
          BLevel := Param;
          BaseAddr := VarAddr;
          BLevel := Param;
end;
end;
end;

PushBaseParamAddr: begin
    IncSP;
    with CStack[SP] do
    begin
        BaseAddr := ParamAddr;
        BLevel := Param;
    end;
end;

PushVal: begin
    {Param is pointer into type table}
    ExpandCC(SP);
    MultlnConstant(SP);
    AddBaseAddr(SP);
    if CStack[SP].IsReal then AddlnConstant(SP)
    else ClearAddIns(SP);
    TDesc := TypeTab1^[Param].TypeDesc;
    if TDesc=SubrangeType then
        TDesc := TypeTab1^[SubTab1^[Param].SubType].TypeDesc;
    case TDesc of
        IntType,
        WordType: begin
            if CStack[SP].IsReal then begin
                CListOutCfmove.w fp'+TopStr+','-(a7)' ,2,
                $f227,$7000+nTop*$80,0,0);
            end else begin
                if CStack[SP].SType = ConstOnly then begin
                    CVal := GBTrunc(CStack[SP].ConstVal);
                    str(CVal,CStr);
                    CMove(W,'#'+CStr,'-(a7)');
                end else begin
                    CMove(W,'d'+TopStr,'-(a7)');
                end;
            end;
        end;
        ShortintType,
        ByteType,
        CharType,
        BoolType,
        EnumType : begin
            if CStack[SP].IsReal then begin
                CSub(L,'#2','a7');
                CListOut('fmove.b fp'+TopStr+','-(a7)' ,2,
                $f217,$7800+nTop*$80,0,0);
            end else begin
                if CStack[SP].SType = ConstOnly then begin
                    CVal := GBTrunc(CStack[SP].ConstVal)
                      and $ff;
                    str(CVal,CStr);
                    CMove(B,'#'+CStr,'-(a7)');
                end else begin
                    CMove(B,'d'+TopStr,'-(a7)');
                end;
            end;
end;
CSub(L,'#2','a7');
CMove(B,'d'+TopStr,'(a7)');
end;
end;
end;

LongintType,
PtrType : begin
if CStack[SP].IsReal then
  FlagError(-500)
else begin
  if CStack[SP].SType = ConstOnly then begin
    CVal := GBTrunc(CStack[SP].ConstVal);
    str(CVal,CStr);
    CMove(L,'# '+ CStr,'-(a7) ');
  end else CMove(L,'d'+TopStr,'-(a7)');
end;
end;

RealType : begin
  AddlnConstant(SP);
  if not CStack[SP].IsReal then i2fpTop;
  CListOutC'fmove.s fp'+TopStr+' -(a7)',2,
  $f227,$6400+nTop*$80,0,0);
end;

DoubleType: begin
  AddlnConstant(SP);
  if not CStack[SP].IsReal then i2fpTop;
  CListOutC'fmove.d fp'+ TopStr+' -(a7)',2,
  $f227,$7400+nTop*$80,0,0);
end;
else FlagError(278);
endSP;
end;

PushVar: begin
  MultlnConstant(SP);
  GetFullAddr(SP);
  PushEA;
  decSP;
end;

Fetch : begin
  {Param is pointer into type table}
  ExpandCC(SP);
  MultlnConstant(SP);
  GetFullAddr(SP);
  TDesc := TypeTabl^ [Param].TypeDesc;
  if TDesc=SubrangeType then
    TDesc := TypeTabl^ [TypeTabl^ [Param].SubType].TypeDesc;
  if TDesc in [ByteType,WordType,CharType,EnumType] then begin
    if posC'd' .AddrStr) <> 0 then begin
      CLea(AddrStr,'a0');
      AddrStr := '(a0)';
EAMode := 2;
EAReg := 0;
NumExtWords := 0;
end;
end;
if TDesc <> StringType then begin
decSP;
incSP;
if TDesc in [ByteType,WordType,CharType,EnumType] then
  CClr(L,'d'+TopStr);
  CStack[SP].SType := RegOnly;
end;
case TDesc of
  LongintType,
  PtrType : CMove(L,AddrStr,'d'+TopStr);
  IntType,
  WordType : CMove(W,AddrStr,'d'+TopStr);
  ShortintType,
  ByteType,
  CharType,
  EnumType : CMove(B,AddrStr,'d'+TopStr);
  BoolType : begin
    CListOut(' tst.b '+ AddrStr.l + NumExtWords,
             + $4a00 + $08*EAMode + EAReg,Ext1,Ext2,Ext3);
    CStack[SP].SType := CondC;
    CStack[SP].CondField := CondNE;
  end;
  RealType :
  begin
    CListOut(' ftnove.s '+AddrStr+',fp'+TopStr,
             $f200 + $8*EAMode + EAReg,
             $4400 + $80*nTop,Ext1,Ext2);
    if 2 + NumExtWords > 4 then
      CListOutC',2+ NumExtWords-4,Ext3,0,0,0);
    CStack[SP].IsReal := true;
  end;
  DoubleType : begin
  CListOut(' fmove.d '+AddrStr+',fp'+TopStr,
            $f200 + $8*EAMode + EAReg,
            $5400 + $80*nTop,Ext1,Ext2);
    if NumExtWords > 2 then
      CListOutC',2+ NumExtWords-4,Ext3,0,0,0);
    CStack[SP].IsReal := true;
  end;
  StringType : FlagError(-880);
  else FlagError(1273);
end;
if TDesc=IntType then
  CListOut('ext.l d'+TopStr,1,$48c0 + nTop,0,0,0);
if TDesc=ShortintType then
  CListOut('extb.l d'+TopStr,1,$49c0 + nTop,0,0,0);
Store : begin
  ExpandCC(SP);
  MultlnConstant(SP);
  AddBaseAddr(SP);
  ClearAddIns(SP);
  GetFullAddr(SPMl);
  {Param points into the type table at entry for variable}
  TDesc := TypeTabl^[Param].TypeDesc;
  if TDesc=SubrangeType then
    TDesc := TypeTabl^[TypeTabl^[Param].SubType].TypeDesc;
  case TDesc of
    IntType,
    WordType: begin
      if CStack[SP].IsReal then begin
        AddlnConstant(SP);
        CListOut(’fmove.w fp’+TopStr+’,’;
           AddrStr,2+NumExtWords,
           $f200 + $8*EAMode + EAReg,
           $7000 + $80*nTop,Ext1,Ext2);
        if NumExtWords > 2 then
          CListOut(’,2+NumExtWords-4,Ext3,0,0,0);
      end else begin
        if CStack[SP].SType=ConstOnly then begin
          CVal := GBTrunc(CStack[SP].ConstVal);
          str(CVal,CStr);
          CMove(W,’#’+CStr, AddrStr);
        end else
          CMove(W,’d’+TopStr, AddrStr);
      end;
  end;
  ShortintType,
  ByteType,
  CharType,
  BoolType,
  EnumType : begin
    if CStack[SP].IsReal then begin
      AddlnConstant(SP);
      CListOut(’fmove.b fp’+TopStr+’,’;
         AddrStr,2+NumExtWords,
         $f200 + $8*EAMode + EAReg,
         $7800 + $80*nTop,Ext1,Ext2);
      if NumExtWords > 2 then
        CListOut(’,2+NumExtWords-4,Ext3,0,0,0);
    end else begin
      if CStack[SP].SType=ConstOnly then begin
        CVal := GBTrunc(CStack[SP].ConstVal) and
               $ff;
        str(CVal,CStr);
        CMove(B,’#’+CStr,AddrStr);
      end else
        CMove(B,’d’+TopStr, AddrStr);
    end;
end;

LongintType,

PtrType : begin
  if CStack[SP].IsReal then begin
    AddInConstant(SP);
    CListOut('fmove.l fp'+TopStr+',',
      + AddrStr,2+NumExtWords,
      $f200 + $8*EAMode + EAReg,
      $6000 + $80*nTop,Ext1,Ext2);
    if NumExtWords > 2 then
      CListOut(',2+NumExtWords-4,Ext3,0,0,0);
  end else begin
    if CStack[SP].SType=ConstOnly then begin
      CVal := GBTrunc(CStack[SP].ConstVal);
      str(CVal,CStr);
      CMove(L,'d'+TopStr,AddrStr);
    end else CMove(L,'d'+TopStr,AddrStr);
  end;
end;

RealType : begin
  AddInConstant(SP);
  if not CStack[SP].IsReal then i2fpTop;
  CListOut('fmove.s fp'+TopStr+',',
    + AddrStr,2+NumExtWords,
    $f200 + $8*EAMode + EAReg,
    $7400 + $80*nTop,Ext1,Ext2);
  if NumExtWords > 2 then
    CListOut(',2+NumExtWords-4,Ext3,0,0,0);
end;

DoubleType: begin
  AddInConstant(SP);
  if not CStack[SP].IsReal then i2fpTop;
  CListOut('fmove.d fp'+TopStr+',',
    + AddrStr,2+NumExtWords,
    $f200 + $8*EAMode + EAReg,
    $7400 + $80*nTop,Ext1,Ext2);
  if NumExtWords > 2 then
    CListOut(',2+NumExtWords-4,Ext3,0,0,0);
end;

StringType : Emit(StoreStr,StringTypePtr);
else FlagError(2273);
end;

decSP;
decSP;
end;

PushStruct : begin
  {Param contains size of the structure to be copied}
  MultiInConstant(SP);
  GetFullAddr(SP);
  CLea(AddrStr,'aO ');
  str((Param+l)and $fffffffe,CStr);
CSub(L,#'+CStr,'a7');
CMove(L,'a7','a1');
if Param <= 32 then begin
  while Param >= 4 do begin
    CMove(L,'(a0)+','(a1)++');
    dec(Param,4);
  end;
  while Param >= 1 do begin
    CMove(B,'(a0)+','(a1)++');
    dec(Param);
  end;
end else begin
  if SP >= 1 then CMove(L,'d1','a2');
  str(Param div 4,CStr);
  CMove(L,#'+CStr,'d1');
  L1 :=NewLabel;
  PlaceLabel(L1);
  CMove(L,'(a0)+','(a1)++');
  CSub(L,#'1','d1');
  DoBranch(CondNE,L1);
  ForgetLabel(L1);
  if Param mod 4 <> 0 then begin
    Param := Param mod 4;
    while Param > 0 do begin
      CMove(B,'(a0)+','(a1)++');
      dec(Param);
    end;
  end;
  if SP >= 1 then CMove(L,'a2','d1');
end;

decSP;

CopyStruct:  begin
  {Param contains size of the structure to be copied}
  if SPV <> 1 then FlagError(-501);
  MultinConstant(SPm1);
  GetFullAddr(SPm1);
  CLea(AddrStr,'a1');
  MultinConstant(SP);
  GetFullAddr(SP);
  CLea(AddrStr,'a0');
  if Param <= 32 then begin
    while Param >= 4 do begin
      CMove(L,'(a0)+','(a1)++');
      dec(Param,4);
    end;
    while Param >= 1 do begin
      CMove(B,'(a0)+','(a1)++');
      dec(Param,1);
    end;
  end else begin
str(Param div 4,CStr);
CMove(L,'#'+CStr,'d1');
L1 := NewLabel;
PlaceLabel(L1);
CMove(L,'(a0)++'(a1)+');
CSub(L,'#1','d1');
DoBranch(CondNE,L1);
ForgetLabel(L1);
if Param mod 4 <> 0 then begin
  Param := Param mod 4;
  while Param > 0 do begin
    CMove(B,'(a0)+','(a1)+');
    dec(Param);
  end;
end;
DecSP;
DecSP;
end;
GetLabelAddress : begin
  { Param is label number }
  str(Param,CStr);
  CListOut('bsr.w *+4;2,$6100,$0002,0,0,0);
  IncSP;
  CStack[SP].SType := RegOnly;
  CListOut('move.l #L'+CStr+'-*,-2,d'+TopStr,3,
          $203c+$200*nTop,0,0,0);
  StoreLabelOffs(Param);
  CAdd(L,'(a7) + ','d'+TopStr);
  CAdd(L,'#2 ','d'+TopStr);
end;
StoreVector : begin
  if SPV <> 1 then FlagError(-502);
  MultInConstant(SPm1);
  AddBaseAddr(SPm1);
  AddInConstant(SPm1);
  ExpandCC(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  CAdd(L,'d'+NextStr,'d'+NextStr);
  CAdd(L,'d'+NextStr,'d'+NextStr);
  CListOut('movec.l vbr,a0',2,$4e7a,1,$8801,0,0,0);
  CAdd(L,'d'+NextStr,'a0');
  CMove(L,'d'+TopStr,'(a0)');
  decSP;
  decSP;
end;
DecWord : begin
  Param := Param AND $fffh;
  str(Param,CStr);
CListOut('dc.w ' + CStr, 1, LoWord(Param), 0, 0, 0);
end;

DecLong : begin
str(Param, CStr);
CListOut('dc.l '+ CStr, 2, HiWord(Param), LoWord(Param), 0, 0);
end;

DoMove : begin
if CStack[SP].IsReal then FlagError(-550);
ExpandCC(SP);
if SPV <> 2 then FlagError(-503);
MultInConstant(SPm2);
GetFullAddr(SPm2);
CLea(AddrStr,'a0');
MultInConstant(SPm1);
GetFullAddr(SPm1);
CLea(AddrStr,'a1');
MultInConstant(SP);
AddBaseAddr(SP);
ClearAddIns(SP);
if (CStack[SP].SType = ConstOnly) and
(CStack[SP].ConstVal <= 40) then begin
while CStack[SP].ConstVal >= 4 do begin
  CStack[SP].ConstVal := CStack[SP].ConstVal - 4;
  CMove(L,'(a0) + ','(a1) + ');
end;
while CStack[SP].ConstVal >= 1 do begin
  CStack[SP].ConstVal := CStack[SP].ConstVal - 1;
  CMove(B,'(a0) + ','(a1) + ');
end;
end else begin
  AddlnConstant(SP);

L1 :=NewLabel;
L2 :=NewLabel;
PlaceLabel(L1);
CCmp(L,'#4','d'+TopStr);
DoBranch(CondLT,L2);
CMove(L,'(a0) + ','(a1) + ');
CSub(L,'#1','d'+TopStr);
DoBranch(CondT,L1);
PlaceForgetLabel(L2);
ForgetLabel(L1);

L1 :=NewLabel;
L2 :=NewLabel;
PlaceLabel(L1);
CCmp(L,'#1','d'+TopStr);
DoBranch(CondLT,L2);
CMove(B,'(a0) + ','(a1) + ');
CSub(L,'#1','d'+TopStr);
DoBranch(CondT,L1);
PlaceForgetLabel(L2);
ForgetLabel(L1);
end;
decSP; decSP; decSP;
end;

DoFillChar : begin
if SPV <> 2 then FlagError(-504);

MultInConstant(SPm1);
AddBaseAddr(SPm1);
ClearAddIns(SPm1);
AddInConstant(SPm1);

MultInConstant(SP);
AddBaseAddr(SP);
ClearAddIns(SP);
AddInConstant(SP);

MultInConstant(SPm2);
GetFullAddr(SPm2);
CLea(AddrStr,'a0');

L1 := NewLabel;
L2 := NewLabel;
CCmp(L,'#0','d' + NextStr);
DoBranch(CondEQ,L2);
PlaceLabel(L1);

CMove(B,'d' + TopStrJ ,(aO)+' );
CSub(L,'# 1 ','d'+NextStr);
DoBranch(CondNE,L1);

PlaceForgetLabel(L2);
ForgetLabel(L1);
decSP; decSP; decSP;
end;

CopyParms : begin
{Param is the size of the parameters plus 12}
{the code address is on top of the stack}
{this is followed by the parameters}
{next comes the address of the process variable }
{finally is the address of the previous process
variable in this PARALLEL statement, which is stored
into the current process variable as a link}
CVal := param - 12 + 4;
str(CVal,CStr);
if CVal > 32000 then FlagError(-222);
ListOut("* Get the process variable address into a0");
CMove(L,CStr+'
(a7)',
'a0');
{test to make sure the process hasn't already been used}
L1 := NewLabel;
CListOut('tst.w (a0),1,$4a50,0,0,0);
DoBranch(CondEQ,L1);
DoMoveQ(1,2); {error number}
OSFunc(1); {error function}
PlaceForgetLabel(L1);
{mark the process as used}
CMove(W,'#ffff,(a0)');
ListOut('* Get the link address into the variable');
{move the link into the variable}
CVal := Param - 12 + 8;
str(CVal,CStr);
CMove(L,CStr+(a7)+,(a0)+);
ListOut('* Discard the code address into the variable');
CMove(L,(a7)+,(a0)+);
ListOut('* Get other cpu data, etc. into the variable');
OSFunc($51);
CMove(W,'d0',(a0)+);
ListOut('* Copy the Parameters into the variable');
CVal := Param - 12;
str(CVal,CStr);
if CVal <> 0 then begin
  CMove(L,'#'+CStr,'d0');
  CMove(L,'a7','a1');
  L1 := NewLabel;
  PlaceLabel(L1);
  CMove(W,'(a1)++,(a0)++');
  CSub(L,'#2','d0');
  DoBranch(CondNE,L1);
  ForgetLabel(L1);
end;
{"remove" the Parameters from the stack}
CVal := Param - 12;
str(CVal,CStr);
CAdd(L,'#'+CStr,'a7');
{"replace" the last link with the new address}
CMove(L,(a7)++,(a7));
end;
Address2Int:  AddBaseAddr(SP);
FixStack :  if StackSize <> 0 then begin
  str(StackSize,CStr);
  CAdd(L,'#'+CStr,'a7');
  Address2Int : AddBaseAddr(SP);
end;
FixStack :  if StackSize <> 0 then begin
  str(StackSize,CStr);
end;
else FlagError(-504);
end;
end.
unit emitpar;
interface
uses gencode,types,tables,errors,list,strstore,options,labels,CircRef;
procedure EmitParInstr(instr:OpCode;Param:longint);
implementation

procedure EmitParInstr(instr:OpCode;Param:longint);
var NumStr:  string[8];
begin
  case instr of
    ParallelCall : begin
      CMove(L,'(a7) +',a0');
      OSFunc($53);
    end;
    StartProcess : begin
      {Param=size of local variables}
      str(Param,NumStr);
      CSub(L,'# '+NumStr,a7);  
      CMove(L,a7,a6); {just in case}
    end;
    EndProcess : begin
      {address of beginning of process is on the cstack}
      MultInConstant(SP);
      GetFullAddr(SP);
      CLea(AddrStr,a0);
      decSP;
      OSFunc(2);
    end;
    StartRsrcProc : begin
      { Param contains local variable size.}
      if Param <> 0 then begin
        str(Param,NumStr);
        CSub(L,'# '+NumStr,a7);
      end;
      CMove(L,a5-(a7));
      CMove(L,a7,a5);
    end;
    EndRsrcProc : begin
      OSFunc($21);
    end;
    CallRsrcProc : begin
      MultInConstant(SP);
      GetFullAddr(SP);
      CLea(AddrStr,a0');
      decSP;
      OSFunc($20);
    end;
    else FlagError(-32);
  end;
end;
unit emitfile;
interface
uses gencode,types,tables,errors,list,strstore,options,labels,CircRef;
procedure EmitFileInstr(instr:OpCode;Param:longint);

implementation

procedure EmitFileInstr(instr:OpCode;Param:longint);
var CStr : string[25];
 Tmplnt : longint;
 SkipLabl : longint;
 i : byte;
 DOSaved,
 DoFlush : boolean;
begin
 case instr of
AssignFile : begin
 if SPV <> 1 then FlagError(-545);
 MultlnConstant(SPm1);
 GetFullAddr(SPm1);
 CLea(AddrStr,'a0');
 if CheckOption('l') then DoMoveQ(2,1)
 else DoMoveQ(2,0);
 if TypeTabl[Param].TypeDesc=TextType then DoMoveQ(1,0)
 else begin
 Tmplnt := TypeTabl[Param].SubType.
   TypeSize;
 str(Tmplnt,CStr);
 CMove(L,'# '+CStr,'d 1');
 OSFunc($10);
 DecSP;
 DecSP;
 CAdd(L,'#256','a7');
 end;
 ResetFile,
 RewriteFile,
 CloseFile : begin
 if (TypeTabl[Param].TypeDesc=ChannelType) and
   (instr=CloseFile) then begin
 if (SPV <> 1) then FlagError(-546);
 MultlnConstant(SPm1);
 GetFullAddr(SPm1);
 CLea(AddrStr,'a0');
 MultlnConstant(SP);
 GetFullAddr(SP);
 CLea(AddrStr,'a1');
 DecSP; DecSP;
 Tmplnt := TypeTabl[Param].SubType.
   TypeSize;
 str(Tmplnt,CStr);

end.

CMove(L,'#'+CStr,'d1');
end else begin
  if (SPV <> 0) then FlagError(-546);
  MultiConstant(SP);
  GetFullAddr(SP);
  CLea(AddrStr,'a0');
  decSP;
end;
if TypeTabl^[Param].TypeDesc=ChannelType then begin
  CMove(L,'(a0)7a0');
  case instr of
    CloseFile : OSFunc($32);
    ResetFile : OSFunc($33);
    RewriteFile : OSFunc($34);
  end
end else begin
  if CheckOption('l') then DoMoveQ(2,1)
  else DoMoveQ(2,0);
  case instr of
    CloseFile : OSFunc($c);
    ResetFile : OSFunc($a);
    RewriteFile : OSFunc($b);
  end;
end;
SeekFile : begin
  if SPV <> 1 then FlagError(-547);
  MultiInConstant(Spm1);
  GetFullAddr(Spm1);
  CLea(AddrStr,'a0');
  if CStack[SP].BaseAddr <> NoBase then FlagError(-548);
  ExpandCC(SP);
  MultiConstant(SP);
  AddInConstant(SP);
  if nTop <> 1 then CMove(L,'d' + TopStr,'d1');
  if CheckOption('l') then DoMoveQ(2,1)
  else DoMoveQ(2,0);
  OSFunc($d);
  DecSP;
  DecSP;
end;
SelectStdIO : CMove(L,'#-1','-(a7)');
SelectIO : begin
  {if SPV <> 0 then FlagError(-549);}  
  ExpandCC(SP);
  MultiConstant(SP);
  GetFullAddr(SP);
  PushEA;
  DecSP;
end;
ReleaseIO : CAdd(L,'#4','a7');
ClrInputLine : begin
  if SPV <> -1 then FlagError(-550);
  CMove(L,'(a7)',"a1");
  OSFunc($13);
end;

PrintInt : begin
  if SPV <> 1 then FlagError(-551);
  ExpandCC(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  {nTop >= 1, nNext >= 0, nNext < nTop}
  {save the field width on the Stack}
  CMove(L,'d'+TopStr,"(a7)");
  {get the integer to be printed into d2}
  if nNext <> 2 then CMove(L,'d'+NextStr,"d2");
  {get nTop back into d1}
  CMove(L,"(a7)+","d1");
  CMove(L,"(a7)","a1");
  OSFunc($14);
  DecSP;
  DecSP;
end;

PrintReal : begin
  if SPV <> 2 then FlagError(-552);
  if not CStack[SPm2].IsReal then FlagError(-553);
  ExpandCC(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  CMove(L,"(a7)","a1");
  str(SPm2,CStr);
  if nTop <> 2 then CMove(L,"d'+TopStr,"d2");
  if nNext <> 1 then CMove(L,"d'+NextStr,"d1");
  CListOut('fmove.d fp+CStr+',,(a7),2,#227,$7400,0,0);
  OSFunc($15);
  CAdd(L,"#8","a7");
  DecSP;
  DecSP;
  DecSP;
end;

PrintChar : begin
if SPV <> 1 then FlagError(-554);
if CStack[SPm1].IsReal then FlagError(-555);
ExpandCC(SP);
MultlnConstant(SP);
AddBaseAddr(SP);
AddlnConstant(SP);
MultlnConstant(SPm1);
AddBaseAddr(SPm1);
AddlnConstant(SPm1);
{save the field width on the stack}
CMove(L,'d'+TopStr,'(a7)');
{get the character into d2}
if nNext <> 2 then CMove(L,'d'+NextStr,'d2');
{get the field width into d1}
CMove(L,'(a7)+','d1');
CMove(L,'(a7)','a1');
OSFunc($16);
DecSP;
DecSP;
end;
PrintString: begin
if SPV <> 1 then FlagError(-556);
if CStack[SPm1].IsReal then FlagError(-557);
ExpandCC(SP);
MultlnConstant(SP);
AddBaseAddr(SP);
AddlnConstant(SP);
if nTop <> 1 then CMove(L,'d'+TopStr,'d1');
CMove(L,'256(a7)','a1');
OSFunc($17);
CAdd(L,'#256Va7');
DecSP;
DecSP;
end;
ReadVar: begin
{Param is pointer into type table}
if SPV <> 0 then FlagError(-558);
ExpandCC(SP);
MultlnConstant(SP);
GetFullAddr(SP);
CLea(AddrStr,'a0');
TDesc := TypeTabl^[Param].TypeDesc;
case TDesc of
LongintType,WordType,
IntType,ByteType,
ShortintType,
SubrangeType: begin
CMove(L,'(a7)','a1');
OSFunc($18);
if CheckOption(R') and
(TDesc <> LongintType) then begin


str(TypeTabl^[Param].p1,CStr);
CCmp(L,'#'+CStr,'d0');
CLListOut('trapl',1,1,$5dfc,0,0,0);
str(TypeTabl^[Param].p2,CStr);
CCmp(L,'#'+CStr,'d0');
CLListOut('trapgt',1,1,$5efc,0,0,0);
end;
if TDesc=SubrangeType then
  TDesc := TypeTabl^[TypeTabl^[Param].
          SubType].TypeDesc;
case TDesc of
    LongintType :  CMove(L,'d07(a0)');
    WordType,
    IntType :  CMove(W,'d07(a0)');
    ByteType,
    ShortintType :  CMove(B,'d07(a0)');
    else FlagError(799);
  end;
end;
RealType,
DoubleType :  begin
  CMove(L,'(a7)','a1');
  CSub(L,'#87a7','a7');
  OSFunc($1b);
  CListOut(fmmove.d (a7),fp0,2,$f21f,$5400,
          0,0);
  if TDesc=RealType then
    CListOut(fmmove.s fp0,(a0),2,$f210,$4400,0,0)
  else
    CListOut(fmmove.d fp0,(a0),2,$f210,$7400,0,0);
end;
CharType :  begin
  CMove(L,'(a7)','a1');
  OSFunc($19);
end;
StringType : begin
  Tmplnt := = TypeTabl^[Param].p1;
  str(Tmplnt,CStr);
  CMove(L,'#'+CStr,'d1');
  CMove(L,'(a7)','a1');
  OSFunc($1a);
end;
DecSP;
end;
PrintEol : begin
  if SPV <> -1 then FlagError(-559);
  DoMoveQ(1,$ff);
  CMove(L,'(a7)','a1');
  DoMoveQ(2,13);
  OSFunc($16);
DoMoveQ(1,$ff);
CMove(L,'(a7)','(a1');
DoMoveQ(2,10);
OSFunc($16);
end;

ReadFile,
WriteFile : begin
  if SPV <> 1 then FlagError(-560);
  MultlnConstant(SPm1);
  GetFullAddr(SPm1);
  CLea(AddrStr,'a0');
  ExpandCC(SP);
  MultlnConstant(SP);
  GetFullAddr(SP);
  CLea(AddrStr,'a1');
  Tmplnt := TypeTabl^[TypeTabl^[Param].SubType].
    TypeSize;
  str(Tmplnt,CStr);
  CMove(L,'# '+ CStr.'d l');
  if TypeTabl^[Param].TypeDesc=ChannelType then begin
    case instr of
      ReadFile : OSFunc($35);
      WriteFile : OSFunc($36);
    end
  end else begin
    if CheckOption(T) then DoMoveQ(2,1)
    else DoMoveQ(2,0);
    case instr of
      ReadFile : OSFunc($1c);
      WriteFile : OSFunc($1d);
    end;
  end;
DecSP;
end;

ChkEOF : begin
  incSP;
  CStack[SP].SType := RegOnly;
  DOSaved := SP <> 0;
  if DOSaved then CMove(L,'d0','(a7'));
  CMove(L,'(a7)','(a1');
  OSFunc($30);
  if DOSaved then begin
    CMove(L,'d0',d'+TopStr);
    CMove(L,'(a7) + ',d0');
  end;
end;

ChkEOC : begin
  MultlnConstant(SP);
  GetFullAddr(SP);
CLea(AddrStr,'a0');
CMove(L,'(a0)',a0); decSP;
incSP;
DOSaved := SP <> 0;
if DOSaved then CMove(L,'d0',-(a7));
CStack[SP].SType := RegOnly;
OSFunc($31);
if DOSaved then begin
  CMove(L,'d0',d'+TopStr);
  CMove(L,'(a7)+','d0');
end;
end;

ChkSelect : begin
{Param is the label number of the statement corresponding to the channel being checked. If the channel is ready,
  push the address of this label onto the stack.
  Otherwise, do nothing.}
MultInConstant(SP);
GetFullAddr(SP);
CLea(AddrStr,'a0');
CMove(L,'(a0)',a0); decSP;
OSFunc($37);
CCmp(L,'#1','d0');
SkipLabl := NewLabel;
DoBranch(CondNE,SkipLabl);
str(Param,CStr);
CListOutCbsr.w *+4,2,$6100,$0002,0,0,0);
CListOut('move1 #L+CStr+1*2,d0',3,$203c,0,0,0); StoreLabelOffs(Param);
CAdd(L,'(a7)+','d0');
CAdd(L,'#2','d0');
CMove(L,'d0','-{(a7)}');
PlaceForgetLabel(SkipLabl);
end;

SkipElse : begin
  str(Param,CStr);
  CListOut('bsr l L'+CStr,3,$61ff,0,0,0);
  StoreLabelOffs(Param);
end;

PushLabelAddr : begin
{Param is the label number}
str(Param,CStr);
CListOut('bsr w *+4,2,$6100,$0002,0,0,0));
CListOut('move1 #L+CStr+1*2,d0',3,$203c,0,0,0); StoreLabelOffs(Param);
CAdd(L,'(a7)+','d0');
CAdd(L,'#2','d0');
CMove(L,'d0','-{(a7)}');
end;
ChooseSelect : begin
  OSFunc($38);
  CListOut('rts',1,$4e75,0,0,0);
end;
FlushIO : OSFunc(9);
else FlagError(-561);
end;
end;
end.
unit emitlog;
interface
uses gencode,types,tables,errors,list,strstore,options,labels,CircRef;
procedure EmitLogCmp(instr:OpCode;Param:longint);

implementation

procedure EmitLogCmp(instr:OpCode;Param:longint);
var
  i : integer;
  RConst : double;
  RSize : longint;
  CVal : longint;
  CStr : string[25];
  BoolResult : longint;
  LDone : longint;
begin
  case instr of
    LogAnd,
    LogOr,
    LogXor: begin
    if CStack[SP].BaseAddr<>NoBase then FlagError(-564);
    if CStack[SPm1].BaseAddr<>NoBase then FlagError(-565);
    ExpandCC(SP);
    if CStack[SP].STYPE = ConstOnly then begin
    if CStack[SPm1].STYPE = ConstOnly then begin
      case instr of
        LogAnd : CStack[SPm1].ConstVal :=
          GBTTrunc(CStack[SP].ConstVal) AND
          GBTTrunc(CStack[SPm1].ConstVal);
        LogOr : CStack[SPm1].ConstVal :=
          GBTTrunc(CStack[SP].ConstVal) OR
          GBTTrunc(CStack[SPm1].ConstVal);
        LogXor : CStack[SPm1].ConstVal :=
          GBTTrunc(CStack[SP].ConstVal) XOR
          GBTTrunc(CStack[SPm1].ConstVal);
      end;
    end else begin
      MultInConstant(Spm1);
      AddInConstant(Spm1);
      CVal := GBTTrunc(CStack[SP].ConstVal);
    end;
    end;
  end;
end.
end.

str(CVal,CStr);
case instr of
  LogAnd : CLListOut('and.l  #'+CStr+',d'+NextStr,3, $0280+nNext,HiWord(CVal),LoWord(CVal),0);
  LogOr : CLListOut('or.l   #'+CStr+',d'+NextStr,3, $0080+nNext,HiWord(CVal),LoWord(CVal),0);
  LogXor : CLListOut('xor.l  #'+CStr+',d'+NextStr,3, $0a80+nNext,HiWord(CVal),LoWord(CVal),0);
end;
end;
end else begin
  MultlnConstant(SP);
  AddlnConstant(SP);
  MultlnConstant(SPml);
  AddlnConstant(SPml);
case instr of
  LogAnd : CLListOut('and.l d'+TopStr+',d'+NextStr,l, $080+$200*nNext+nTop,0,0,0);
  LogOr : CLListOut('or.l   d'+TopStr+',d'+NextStr,l, $8080+nTop+nNext*$200,0,0,0);
  LogXor: CLListOut('xor.l  d'+TopStr+',d'+NextStr,l, $0a80 + $200*nTop + nNext,0,0,0);
end;
end;
DecSP;
end;

LogNot: begin
  if CStack[SP].BaseAddr<>NoBase then FlagError(-566);
  if CStack[SP].SType = CondC then begin
    case CStack[SP].CondField of
      CondCC : CStack[SP].CondField := CondCS;
      CondCS : CStack[SP].CondField := CondCC;
      CondEQ : CStack[SP].CondField := CondNE;
      CondGE : CStack[SP].CondField := CondLT;
      CondGT : CStack[SP].CondField := CondLE;
      CondHI : CStack[SP].CondField := CondLS;
      CondLE : CStack[SP].CondField := CondGT;
      CondLS : CStack[SP].CondField := CondHI;
      CondLT : CStack[SP].CondField := CondGE;
      CondMI : CStack[SP].CondField := CondPL;
      CondNE : CStack[SP].CondField := CondEQ;
      CondPL : CStack[SP].CondField := CondMI;
      CondVC : CStack[SP].CondField := CondVS;
      CondVS : CStack[SP].CondField := CondVC;
      CondT : CStack[SP].CondField := CondF;
      CondF : CStack[SP].CondField := CondT;
    end;
  end else begin
    MultlnConstant(SP);
    AddlnConstant(SP);
    CLListOut('neg.l  d'+TopStr,l,$4480+nTop,0,0,0);
CAdd(1,#1',d'+TopStr);
end;
end;

BitwiseNot: begin
  if CStack[SP].BaseAddr<>NoBase then FlagError(-567);
  ExpandCC(SP);
  MultInConstant(SP);
  AddInConstant(SP);
  CListOut('not'+TopStr,'d'+TopStr,1,$4680+nTop,0,0,0);
end;

EndOfOr,
EndOfAnd : begin
  LDone := NewLabel;
  CListOut('ori.b #4,ccr',2,$003c,$0004,0,0);
  DoBranch(CondT,LDone);
  PlaceLabel(Param);
  CListOut('andi.b #0,ccr',2,$023c,$0000,0,0);
  PlaceForgetLabel(LDone);
  IncSP;
  CStack[SP].SType := CondC;
  if instr=EndOfOr then CStack[SP].CondField := CondNE
  else CStack[SP].CondField := CondEQ;
end;

CmpEQ,
CmpNE,
CmpLT,
CmpLE,
CmpGT,
CmpGE : begin
  if not (instr in [CmpEQ,CmpNE]) then begin
    if CStack[SP].BaseAddr<>NoBase then FlagError(-568);
    if CStack[SPm1].BaseAddr<>NoBase then FlagError(-569);
  end;
  SetTypes;
  ExpandCC(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  ClearAddIns(SP);
  MultInConstant(SPm1);
  AddBaseAddr(SPm1);
  ClearAddIns(SPm1);
  if (CStack[SP].SType=ConstOnly) and not CStack[SP].IsReal then begin
    if (CStack[SPm1].SType=ConstOnly) then begin
      BoolResult := 0;
      case instr of
        CmpEQ : if CStack[SPm1].ConstVal = CStack[SP].ConstVal then
          BoolResult := 1;
        CmpNE : if CStack[SPm1].ConstVal <> CStack[SP].ConstVal then
          BoolResult := 1;
        CmpLT : if CStack[SPm1].ConstVal < CStack[SP].ConstVal then
          BoolResult := 1;
      end;
    end;
  end;
CmpLE : if CStack[SPm1].ConstVal <= CStack[SP].ConstVal then
  BoolResult := 1;
CmpGT : if CStack[SPm1].ConstVal > CStack[SP].ConstVal then
  BoolResult := 1;
CmpGE : if CStack[SPm1].ConstVal >= CStack[SP].ConstVal then
  BoolResult := 1;
end;
declSP;
declSP;
incSP;
CStack[SP].SType := CondC;
if BoolResult = 1 then CStack[SP].CondField := 0
else CStack[SP].CondField := 1;
end else begin
  AddInConstant(SPm1);
  SetTypes;
  CVal := GBTrunc(CStack[SP].ConstVal);
  str(CVal,CStr);
  CCmp(L,'# '+ CStr.'d'+ NextStr);
  DecSP;
  DecSP;
  IncSP;
  CStack[SP].SType := CondC;
  case instr of
    CmpEQ: CStack[SP].CondField := CondEQ;
    CmpNE: CStack[SP].CondField := CondNE;
    CmpLT: CStack[SP].CondField := CondLT;
    CmpLE: CStack[SP].CondField := CondLE;
    CmpGT: CStack[SP].CondField := CondGT;
    CmpGE: CStack[SP].CondField := CondGE;
  end;
end;
end else begin
  AddInConstant(SP);
  AddInConstant(SPm1);
  if CStack[SP].IsReal then begin
    CListOut('fcmp.x fp'+TopStr+'+fp'+NextStr,2,$f200,
      $0038+$400*nTop+$80*nNext,0,0);
    DecSP;
    case instr of
      CmpEQ: CListOut('fseq.b d'+TopStr,2,$f240+nTop,$0001,0,0);
      CmpNE: CListOut('fsne.b d'+TopStr,2,$f240+nTop,$000e,0,0);
      CmpLT: CListOut('fslt.b d'+TopStr,2,$f240+nTop,$0014,0,0);
      CmpLE: CListOut('fsle.b d'+TopStr,2,$f240+nTop,$0015,0,0);
      CmpGT: CListOut('fsgt.b d'+TopStr,2,$f240+nTop,$0012,0,0);
      CmpGE: CListOut('fsge.b d'+TopStr,2,$f240+nTop,$0013,0,0);
    end;
    CListOut('neg.b d'+TopStr,1,$4400+nTop,0,0,0);
  CStack[SP].IsReal := false;
    CListOut('extb.l d'+TopStr,1,$49c0+nTop,0,0,0);
  end else begin
  end;
end;
CCmp(L,'d'+TopStr,'d'+NextStr);
DecSP;
DecSP;
IncSP;
CStack[SP].SType := CondC;
case instr of
CmpEQ: CStack[SP].CondField := CondEQ;
CmpNE: CStack[SP].CondField := CondNE;
CmpLT: CStack[SP].CondField := CondLT;
CmpLE: CStack[SP].CondField := CondLE;
CmpGT: CStack[SP].CondField := CondGT;
CmpGE: CStack[SP].CondField := CondGE;
end;
end;
end;
end.

end.
unit emitstr;
interface
uses gencode.types.tables.errorsjist.strstore,options,labels,CircRef;
procedure EmitStrInstr(instr:OpCode;Param:longint);

implementation
var
FetchStrLabel,
StoreStrLabel,
AddStrStrLabel,
AddStrCharLabel,
AddCharStrLabel : longint;

procedure EmitStrInstr(instr:OpCode;Param:longint);
var LablName : string[25];
i : integer;
RConst : double;
RSize : longint;
CVal : longint;
CStr : string[25];
TmpInt : longint;
SConst : DataString;
L1,L2,L3,L4,L5,L6 : longint;
LStr1,LStr2 : string[8];
begin
  case instr of
  ChkStrIndex : begin
    ExpandCC(SP);
    {Param is maximum string length}
    if CheckOption('R') then begin

ClearAddIns(SP);
if CStack[SP],SType=ConstOnly then begin
    if (CStack[SP],ConstVal < 0) or (CStack[SP],ConstVal
    > Param) then FlagError(214);
end else begin
    AddInConstant(SP);
    CListOut('tst.'d'+TopStr,'$4a80+nTop,0,0,0);
    CListOut('trapl't,'$5dfc,0,0,0);
    str(Param,CStr);
    CVal := $1000000*ord(SConst[Tmplnt]) +
            $10000*ord(SConst[Tmplnt+1]) +
            $100*ord(SConst[Tmplnt+2]) +
            ord(SConst[Tmplnt+3]);
    dec(RSize,4);
    str(CVal,CStr);
    CMove(L,'# '+ CStr,' (a0) + ');
end;
end;
StrConstant: begin
    GetStringConstant(SConst,Param);
    if SP > 6 then FlagError(259);
    IncSP;
    CStack[SP],SType := StrOnly;
    CSub(L,'#256','a7');
    CMove(L,'a7','aO ');
    RSize := length(SConst) + 1;
    Tmplnt := 0;
    while RSize >= 4 do begin
        CVal := $1000000*ord(SConst[Tmplnt]) +
                $10000*ord(SConst[Tmplnt+1]) +
                $100*ord(SConst[Tmplnt+2]) +
                ord(SConst[Tmplnt+3]);
        inc(Tmplnt,4);
        dec(RSize,4);
        str(CVal,CStr);
        CMove(L,'# '+ CStr,' (a0) + ');
    end;
    while RSize >= 2 do begin
        CVal := $100*ord(SConst[Tmplnt]) +
                ord(SConst[Tmplnt+1]);
        inc(Tmplnt,2);
        dec(RSize,2);
        str(CVal,CStr);
        CMove(W,'# '+ CStr,'(a0) + ');
    end;
    if RSize = 1 then begin
        CVal := $100*ord(SConst[Tmplnt]);
        str(CVal,CStr);
        CMove(W,'# '+ CStr,'(a0) + ');
    end;
end;
StoreStr: begin
    { Param is a pointer into the type table }
    ExpandCC(SP);
    if SPV <> 1 then FlagError(-536); {fix this}
GetFullAddr(SPm1);
PUSH EA;
if StoreStrLabel = 0 then StoreStrLabel := NewLabel;
str(StoreStrLabel, LStr1);
CListOut('b', L'+LStr1, 3,$61ff,0,0,0);
StoreLabelOffs(StoreStrLabel);
decSP;
decSP;
end;

PushStrValParam : begin
  ExpandCC(SP);
  {Param is pointer into type table}
  DecSP;
end;

PushCharAsStrParam : begin
  ExpandCC(SP);
  AddInConstant(SP);
  CSub(L,'#256','a 7 ');
  CMove(B,'#1','(a 7 )');
  CMove(B,'d 1','(a 7 )');
  DecSP;
end;

FetchStr : begin
  IncSP;
  if SP > 6 then FlagError(259);
  PushEA;
  if FetchStrLabel = 0 then FetchStrLabel := NewLabel;
  str(FetchStrLabel, LStr1);
  CListOut('b', L'+LStr1, 3,$61ff,0,0,0);
  StoreLabelOffs(FetchStrLabel);
  IncSP;
  CStack[SP].SType := StrOnly;
end;

StoreCharInStr : begin
  if SPV <> 1 then FlagError(-537); {fix this}
  ExpandCC(SP);
  MultInConstant(SP);
  AddBaseAddr(SP);
  AddInConstant(SP);
  GetFullAddr(Spm1);
  if CStack[SP].IsReal then FlagError(-538);
  if AddrStr <> '('a0') then CLea(AddrStr,'a0');
  CMove(B,'#1','(a0) +');
  CMove(B,'d 1','(a0)');
  DecSP;
  DecSP;
end;

AddStrings : begin
  if AddStrStrLabel = 0 then AddStrStrLabel :=NewLabel;
str(AddStrStrLabel, LStrl);
CListOut(' bsr.1 L' + LStrl, 3, $61ff, 0, 0, 0);
StoreLabelOffs(AddStrStrLabel);
DecSP;
end;
AddCharStr : begin
  ExpandCC(SP);
  if CStack[SPml].BaseAddr<>NoBase then FlagError(-539);
  MultInConstant(SPml);
  AddInConstant(SPml);
  CMove(L, 'd' + NextStr, '- (a7) ');
  if AddCharStrLabel=0 then AddCharStrLabel := NewLabel;
  str(AddCharStrLabel, LStrl);
  CListOut(' bsr.1 L' + LStrl, 3, $61ff, 0, 0, 0);
  StoreLabelOffs(AddCharStrLabel);
  DecSP;
  CStack[SP].SType := StrOnly;
end;
AddStrChar : begin
  if CStack[SP].BaseAddr< >NoBase then FlagError(-540);
  ExpandCC(SP);
  MultInConstant(SP);
  AddInConstant(SP);
  CMove(L, 'd' + TopStr, '- (a7) ');
  if AddStrCharLabel = 0 then AddStrCharLabel := NewLabel;
  str(AddStrCharLabel, LStrl);
  CListOut(' bsr.1 L' + LStrl, 3, $61ff, 0, 0, 0);
  StoreLabelOffs(AddStrCharLabel);
  DecSP;
end;
AddCharChar : begin
  if CStack[SP].BaseAddroNoBase then FlagError(-541);
  ExpandCC(SP);
  MultInConstant(SP);
  AddInConstant(SP);
  if CStack[SPml].BaseAddroNoBase then FlagError(-542);
  MultInConstant(SPml);
  AddInConstant(SPml);
  CSub(L, '#256', 'a7');
  CMove(B, '#2', '(a7)');
  CMove(B, 'd' + NextStr, '{a7}');
  CMove(B, 'd' + TopStr, '{a7}');
  DecSP;
  CStack[SP].SType := StrOnly;
end;
CharToStr : begin
  if CStack[SP].BaseAddroNoBase then FlagError(-543);
  MultInConstant(SP);
  AddInConstant(SP);
  CSub(L, '#256', 'a7');
  CMove(B, '#1', '(a7)');
CMove(B,'d'+TopStr,'1(a7)');
DecSP;
IncSP;
CStack[SP].SType := StrOnly;
end;

ExpandFuncStr : begin
  {Param is length of string}
  {this is a nop, since all strings are now 255 chars long}
  end;

CmpStrs : begin
  L1 := NewLabel;
  L2 := NewLabel;
  L3 := NewLabel;
  L4 := NewLabel;
  L5 := NewLabel;
  L6 := NewLabel;
  ExpandCC(SP);
  CMove(L,'a7', 'a0');
  CAdd(L,'#1', 'a0');
  CMove(L,'d0', 'a1');
  PlaceLabel(L1);
  CListOutCtst.b (a7),1,$4a17,0,0,0);
  DoBranch(CondEQ,L3);
  CListOutCtst.b 256(a7),2,$4a2f,256,0,0);
  DoBranch(CondEQ,L3);
  CSub(B,'#1','(a7)');
  CSub(B,'#1',256(a7));
  CMove(B,'256(a0)','d0');
  CCmp(B,'(a0) + ','d0');
  DoBranch(CondEQ,L1);
  DoBranch(CondCC,L2);
  DoMoveQ(0,3);
  DoBranch(CondCS,L6);
  PlaceLabel(L2);
  DoMoveQ(0,1);
  DoBranch(CondCS,L6);
  PlaceLabel(L3);
  CListOutCtst.b (a7),1,$4a17,0,0,0);
  DoBranch(CondNE,L5);
  CListOutCtst.b 256(a7),2,$4a2f,256,0,0);
  DoBranch(CondEQ,L4);
  DoMoveQ(0,3);
  DoBranch(CondCS,L6);
  PlaceLabel(L4);
  DoMoveQ(0,2);
  DoBranch(CondCS,L6);
  PlaceLabel(L5);
  DoMoveQ(0,1);
  PlaceLabel(L6);
  CAdd(L,'#512', 'a7');
  CCmp(L,'#2', 'd0');
  CMove(L,'a1', 'd0');
decSP;
decSP;
incSP;
case Param of
  1 : {<}  CListOut('slt.b d'+TopStr,1,$5dc0+nTop,0,0,0);
  2 : {<=}  CListOut('sle.b d'+TopStr,1,$5fc0+nTop,0,0,0);
  3 : {=}  CListOut('seq.b d'+TopStr,1,$57c0+nTop,0,0,0);
  4 : {<>}  CListOut('sne.b d'+TopStr,1,$56c0+nTop,0,0,0);
  5 : {>=}  CListOut('sge.b d'+TopStr,1,$5cc0+nTop,0,0,0);
  6 : {>}  CListOut('sgt.b d'+TopStr,1,$5ec0+nTop,0,0,0);
end;

CListOut('neg.b d'+TopStr,1,$4400+nTop,0,0,0);
CListOut('extb.l d'+TopStr,1,$49c0+nTop,0,0,0);

ForgetLabel(L1); ForgetLabel(L2); ForgetLabel(L3);
ForgetLabel(L4); ForgetLabel(L5); ForgetLabel(L6);
end;

CmpStrChar, CmpCharStr : begin
  ExpandCC(SP);
  LI := NewLabel; L2 := NewLabel; L3 := NewLabel;
  L4 := NewLabel;

  if instr = CmpStrChar then L6 := nTop else L6 := nNext;
  str(L6,LStr2);
  CMove(L,'d0Val*);
  CListOut('tst.b (a7)',1,$4a17,0,0,0);
  DoBranch(CondNE,L1);
  DoMoveQ(0,1);
  DoBranch(CondCS,L2);
  PlaceLabel(L1);
  CCmp(B,'(a7)',d'+LStr2);
  DoBranch(CondEQ,L3);
  DoBranch(CondLT,L4);
  DoMoveQ(0,1);
  DoBranch(CondCS,L2);
  PlaceLabel(L3);
  CMove(B,'(a7)',d0);
  CCmp(B,'#1',d0);
  DoBranch(CondNE,L4);
  DoMoveQ(0,2);
  DoBranch(CondCS,L2);
  PlaceLabel(L4);
  DoMoveQ(0,3);
  PlaceLabel(L2);
  CAdd(L,'#256',a7);
  CCmp(L,'#2',d0);
  CMove(L,'a1',d0);
  decSP;
  decSP;
  incSP;
case Param of
1: \{<\} CListOut('slt.b d'+TopStr,l,$5dc0+nTop,0,0,0);
2: \{\le\} CListOut('sle.b d'+TopStr,l,$5fc0+nTop,0,0,0);
3: \{=\} CListOut('seq.b d'+TopStr,l,$57c0+nTop,0,0,0);
4: \{\ge\} CListOut('sne.b d'+TopStr,l,$56c0+nTop,0,0,0);
5: \{\ge\} CListOut('sge.b d'+TopStr,l,$5cc0+nTop,0,0,0);
6: \{=\} CListOut('sgt.b d'+TopStr,l,$5ec0+nTop,0,0,0);

end;
CListOut('neg.b d'+TopStr,1,$4400+nTop,0,0,0);
CListOut('extb.l d'+TopStr,1,$49c0+nTop,0,0,0);

ForgetLabel(L1); ForgetLabel(L2); ForgetLabel(L3);
ForgetLabel(L4);
end;
InitStr : begin
  AddStrStrLabel := 0;
  FetchStrLabel := 0;
  StoreStrLabel := 0;
  AddStrCharLabel := 0;
  AddCharStrLabel := 0;
end;
CloseStr : begin
  if AddStrStrLabel <> 0 then begin
    ListOut("* Routine to add two strings");
    PlaceForgetLabel(AddStrStrLabel);
    CMove(L,'d0','-(a7)');
    CMove(L,'a0','- (a7)');
    CMove(L,'a 1 ( a 7 ) ');
    CLeaC16(a7)','a0' );
    CLeaC272(a7)','a l' );
    CMove (B,'  (aO) ' ,'dO ' ) ;
    CAdd(B,'(a l)','dO ' );
    LI := NewLabel;
    DoBranch (CondCC,LI);
    CMovetB.XaDVdO' );
    CListOutCnot.b dO ',1,$4680,0,0,0);
    CMove (B,'dO ','  (aO) ' );
    PlaceForgetLabel (LI);
    CClr(L,'d0');
    CMove(B,'(a0)+','d0');
    CAdd(L,'d0','a1');
    CMove(B,'(a0)+','d0');
    L1 := NewLabel;
    DoBranch(CondCC,L1);
    CMove(B,'(a1)',d0');
    CListOut('not.b d0',1,$4680,0,0,0);
    CMove(B,'d0','(a0)');
    PlaceForgetLabel(L1);
    CClr(L,'d0');
    CMove(B,'(a1)+','d0');
    CAdd(L,'d0','a1');
    CMove(B,'(a0)+','d0');
    L1 := NewLabel;
    L2 := NewLabel;
    DoBranch(CondEQ,L2);
    PlaceLabel(L1);
    CMove(B,'(a0)+','(a1)+');
    CSub(B,'#1','d0');
DoBranch(CondNE,L1);
PlaceLabel(L2);
ForgetLabel(L1);
ForgetLabel(L2);

CMove(B,'16(a7)','d0');
CAdd(B,'d0','272(a7)');

CMove(L,'(a7) +','a1');
CMove(L,'(a7) +','a0');
CMove(L,'(a7) +','d0');

CMove(L,'(a7)','256(a7)');
CAdd(L,'#256','a7');
CListOut('rts',1,$4e75,0,0,0);
end;
if FetchStrLabel <> 0 then begin
ListOut("* Routine to fetch a string");
PlaceForgetLabel(FetchStrLabel);

CSub(L,'#252','a7');
CMove(L,'252(a7)','(a7)');
CMove(L,'d0',(-(a7)');
CMove(L,'a0',-(a7)');
CMove(L,'a1',-(a7)');
CMove(L,'268(a7)','a0');
CLea('16(a7)','a1');
CMove(L,'#16','d0');

L1 := NewLabel;
PlaceLabel(L1);
CMove(L,'(a0) +','(a1) +');
CMove(L,'(a0) +','(a1) +');
CMove(L,'(a0) +','(a1) +');
CMove(L,'(a0) +','(a1) +');
CSub(L,'#1','d0');
DoBranch(CondNE,L1);
ForgetLabel(L1);

CMove(L,'(a7) +','a1');
CMove(L,'(a7) +','a0');
CMove(L,'(a7) +','d0');
CListOut('rts',1,$4e75,0,0,0);
end;
if StoreStrLabel <> 0 then begin
ListOut("* Routine to store a string");
PlaceForgetLabel(StoreStrLabel);

CMove(L,'d0',(-(a7)');
CMove(L,'a0',-(a7)');
CMove(L,'a1',-(a7)');
CMove(L,'16(a7)', 'a1');
CLEa('20(a7)', 'a0');
CMove(L,'#16', 'd0');

L1 := NewLabel;
PlaceLabel(L1);
CMove(L,'(a0)+', '(a1)');
CMove(L,'(a0)+', '(a1)');
CMove(L,'(a0)+', '(a1)');
CMove(L,'(a0)+', '(a1)');
CSub(L,'#1', 'd0');
DoBranch(CondNE, L1);
ForgetLabel(L1);

CMove(L,'(a7)+', 'a1');
CMove(L,'(a7)+', 'a0');
CMove(L,'(a7)+', 'd0');
CMove(L,'(a7)', '260(a7)');
CAdd(L,'#260', 'a7');
CLListOut('rts', 1, $4e75, 0, 0, 0);
end;
if AddCharStrLabel <> 0 then begin
    ListOut("* Routine to perform string = char + string");
    PlaceForgetLabel(AddCharStrLabel);

    CMove(L,'d0', '-(a7)');
    CMove(L,'a0', -(a7));
    CMove(L,'a1', -(a7));
    CLea(278(a7), 'a1');
    CLea(275(a7), 'a0');
    CMove(L,'#254', 'd0');

    L1 := NewLabel;
    PlaceLabel(L1);
    CMove(B,-(a0), -(a1));
    CSub(L,'#1', 'd0');
    DoBranch(CondNE, L1);
    ForgetLabel(L1);

    CMove(L, '16(a7)', 'd0');
    CMove(B,'d0', '21(a7)');
    CMove(B, '20(a7)', 'd0');
    CAdd(B,'#1', 'd0');
    L1 := NewLabel;
    DoBranch(CondCS, L1);
    CMove(B,'d0', '20(a7)');
    PlaceForgetLabel(L1);

    CMove(L,'(a7)+', 'a1');
    CMove(L,'(a7)+', 'a0');
procedure EmitMathInstr(instr: OpCode; Param: longint);

implementation

procedure EmitMathInstr(instr: OpCode; Param: longint);
var
  i : integer;
  RConst : double;
  RSize : longint;
  ShiftVal,
  ShiftTmp,
CVal : longint;
DoShLabl,DoneLabl : longint;
ShiftStr,
CStr : string[15];

begin
  Simplify;
  case instr of
    IncVar: begin
      MultInConstant(SPm1);
      GetFullAddr(SPm1);
      {Param points into the type table at entry for variable}
      TDesc := TypeTabl^[Param].TypeDesc;
      if TDesc=SubrangeType then
        TDesc := TypeTabl^[TypeTabl^[Param].SubType].TypeDesc;
      if CStack[SP].SType = ConstOnly then begin
        str(GBTrunc(CStack[SP].ConstVal),CStr);
        CStr := '# ' + CStr;
        case TDesc of
          IntType,
          WordType : CAdd(W,CStr,AddrStr);
          ShortintType,
          ByteType : CAdd(B,CStr,AddrStr);
          LongintType : CAdd(L,CStr,AddrStr);
          else FlagError(3273);
        end;
      end else begin
        MultInConstant(SP);
        AddInConstant(SP);
        case TDesc of
          IntType,
          WordType : CAdd(W,'d'+TopStr,AddrStr);
          ShortintType,
          ByteType : CAdd(B,'d'+TopStr,AddrStr);
          LongintType : CAdd(L,'d'+TopStr,AddrStr);
          else FlagError(3273);
        end;
      end else begin
        MultInConstant(SP);
        AddInConstant(SP);
        case TDesc of
          IntType,
          WordType : CAdd(W,'d'+TopStr,AddrStr);
          ShortintType,
          ByteType : CAdd(B,'d'+TopStr,AddrStr);
          LongintType : CAdd(L,'d'+TopStr,AddrStr);
          else FlagError(3273);
        end;
      end;
      decSP;
    end;
    DecVar: begin
      MultInConstant(SPm1);
      GetFullAddr(SPm1);
      {Param points into the type table at entry for variable}
      TDesc := TypeTabl^[Param].TypeDesc;
      if TDesc=SubrangeType then
        TDesc := TypeTabl^[TypeTabl^[Param].SubType].TypeDesc;
      if CStack[SP].SType = ConstOnly then begin
        str(GBTrunc(CStack[SP].ConstVal),CStr);
        CStr := '# ' + CStr;
        case TDesc of
IntType,  
WordType : CSub(W,CStr,AddrStr);
ShortintType,  
ByteType : CSub(B,CStr,AddrStr);
LongintType : CSub(L,CStr,AddrStr);
else FlagError(3273);
end;
end else begin
MultlnConstant(SP);
AddlnConstant(SP);
case TDesc of  
IntType,  
WordType : CSub(W,'d'+TopStr,AddrStr);
ShortintType,  
ByteType : CSub(B,'d'+TopStr,AddrStr);
LongintType : CSub(L,'d'+TopStr,AddrStr);
else FlagError(3273);
end;
end;
decSP;
decSP;
end;
Add: begin
ExpandCC(SP);
if CStack[SP].BaseAddr <> NoBase then FlagError(-505); {fix this}
MultlnConstant(SP);
MultlnConstant(SPm1);
SetTypes;
{ If there's a constant in the top stack entry, propagate it down }  
if CStack[SP].SType in [ConstOnly,ConstPlusReg] then begin
if CStack[SPm1].SType in [ConstOnly,ConstPlusReg] then begin
  CStack[SPm1].ConstVal := CStack[SPm1].ConstVal +
  CStack[SP].ConstVal
end else begin
  CStack[SPm1].ConstVal := CStack[SP].ConstVal;
  case CStack[SPm1].SType of
    Empty : CStack[SPm1].SType := ConstOnly;
    RegOnly : CStack[SPm1].SType := ConstPlusReg;
  end;
end;
end;
if CStack[SP].SType in [RegOnly,ConstPlusReg] then begin
if CStack[SPm1].SType in [RegOnly,ConstPlusReg] then begin
if CStack[SP].IsReal then begin
  CListOut('fadd.x fp'+TopStr+',fp'+NextStr,2,
  $200,$0022 + $400*nTop + $80*nNext,0,0);
end else CAdd(L,'d'+TopStr,'d'+NextStr);
end else begin
if CStack[SP].IsReal then begin
  CListOut('fmove.x fp'+TopStr+',fp'+NextStr,2,

\$200,\$0000 + \$400*nTop + \$80*nNext,0,0);} 

case CStack[SPml].SType of 
  Empty : CStack[SPml].SType := RegOnly; 
  ConstOnly : CStack[SPml].SType := ConstPlusReg; 
end; 
end else begin 
  CStack[SP].BaseAddr := CStack[SPml].BaseAddr; 
  CStack[SP].BLevel := CStack[SPml].BLevel; 
  CStack[SP].ConstVal := CStack[SPml].ConstVal; 
  case CStack[SPml].SType of 
    Empty : CStack[SP].SType := RegOnly; 
    ConstOnly : CStack[SP].SType := ConstPlusReg; 
  end; 
  CStack[SPml].SType := NewDead; 
end; 
end; 
end; 
end; 

decSP; 

Subtract : begin 
  ExpandCC(SP); 
  if CStack[SP].BaseAddr <> NoBase then FlagError(-506); \{fix\} 
  MultinConstant(SP); 
  MultinConstant(SPml); 
  SetTypes; 

  \{ Propagate any constant in the top value downward. } 
  if CStack[SP].SType in [ConstOnly,ConstPlusReg] then begin 
    if CStack[SPml].SType in [ConstOnly,ConstPlusReg] then begin 
      CStack[SPml].ConstVal := CStack[SPml].ConstVal - CStack[SP].ConstVal; 
    end else begin 
      CStack[SPml].ConstVal := -CStack[SP].ConstVal; 
      case CStack[SPml].SType of 
        Empty : CStack[SPml].SType := ConstOnly; 
        RegOnly : CStack[SPml].SType := ConstPlusReg; 
      end; 
    end; 
  end; 
end; 

  \{ Propagate any register value in the top downward. } 
  if CStack[SP].SType in [RegOnly,ConstPlusReg] then begin 
    if CStack[SPml].SType in [RegOnly,ConstPlusReg] then begin 
      CListOut('fsub.x fp'+TopStr+'+,fp'+NextStr,2, 
      \$200,\$0028 + \$400*nTop + \$80*nNext,0,0); 
    end else CSub(L,' d'+TopStr,' d'+NextStr); 
    end else begin 
      CListOut('fmove.x fp'+TopStr+'+,fp'+NextStr,2, 
      \$200,\$0000 + \$400*nTop + \$80*nNext,0,0); 
    end; 
end;
CListOut('fneg.x dp'+NextStr,2,$f200,
$001a + $400*nNext + $80*nNext,0,0);
case CStack[SPml].SType of
  Empty : CStack[SPml].SType := RegOnly;
  ConstOnly : CStack[SPml].SType := ConstPlusReg;
end;
end else begin
  CStack[SP].BaseAddr := CStack[SPml].BaseAddr;
  CStack[SP].BLevel := CStack[SPml].BLevel;
  CStack[SP].ConstVal := CStack[SPml].ConstVal;
  case CStack[SPml].SType of
    Empty : CStack[SP].SType := RegOnly;
    ConstOnly : CStack[SP].SType := ConstPlusReg;
    end;
  CStack[SPml].SType := NewDead;
  CListOut('neg.l d'+TopStr,1,
$4480 + nTop,0,0,0);
  end;
end;
end;
decSP;
end;
Mult : begin
  if CStack[SP].BaseAddr<>NoBase then FlagError(-507);
  if CStack[SPml].BaseAddr<>NoBase then FlagError(-508);
  ExpandCC(SP);
  ClearAddIns(SP);
  ClearAddIns(SPml);
  SetTypes;

  { Propagate any constants downward. }
  if CStack[SP].SType in [ConstOnly,ConstTimesReg] then begin
    case CStack[SPml].SType of
      Empty : FlagError(-509);
      ConstPlusReg : FlagError(-510);
      ConstTimesReg,
      ConstOnly : CStack[SPml].ConstVal := CStack[SP].ConstVal * 
                 CStack[SPml].ConstVal;
      RegOnly : begin
        CStack[SPml].ConstVal := CStack[SP].ConstVal;
        CStack[SPml].SType := ConstTimesReg;
        end;
    end;
  end;

  { Propagate any registers downward. }
  if CStack[SP].SType in [RegOnly,ConstTimesReg] then begin
    case CStack[SPml].SType of
      Empty : FlagError(-511);
      ConstPlusReg : FlagError(-512);
      ConstOnly : begin

if CStack[SP].IsReal then begin
  CListOut('fmove.x fp'+TopStr+' ,fp'+NextStr,2,
    $f200, $0000 + $400*nTop + $80*nNext,0,0);
  CStack[SPm1].SType := ConstTimesReg;
end else begin
  CStack[SP].BaseAddr := CStack[SPm1].BaseAddr;
  CStack[SP].BLevel := CStack[SPm1].BLevel;
  CStack[SP].ConstVal := CStack[SPm1].ConstVal;
  CStack[SP].SType := NewDead;
end;

ConstTimesReg : begin
  if CStack[SP].IsReal then begin
    CListOut('fmul.x fp'+TopStr+' ,fp'+NextStr,2,$f200,
      $0023 + $400*nTop + $80*nNext,0,0);
  end else begin
    CListOut('mulsl d'+TopStr+' ,d'+NextStr,2,$4c00 + nTop,
      $0800 + $1000*nNext,0,0);
  end;
end;

RegOnly : begin
  if CStack[SP].IsReal then begin
    CListOut('fmul.x fp'+TopStr+' ,fp'+NextStr,2,$f200,
      $0023 + $400*nTop + $80*nNext,0,0);
  end else begin
    CListOut('mulsl d'+TopStr+' ,d'+NextStr,2,$4c00 + nTop,
      $0800 + $1000*nNext,0,0);
  end;
end;

DecSP;
end;

Divide : begin
  ExpandCC(SP);
  if CStack[SP].BaseAddr<>NoBase then FlagError(-513);
  if CStack[SPm1].BaseAddr<>NoBase then FlagError(-514);
  if (CStack[SP].SType = ConstOnly) and
    (CStack[SPm1].SType = ConstOnly) then begin
    if CStack[SP].ConstVal = 0.0 then FlagError(300);
    CStack[SPm1].ConstVal := CStack[SPm1].ConstVal
      / CStack[SP].ConstVal;
  decSP;
  CStack[SP].IsReal := true;
end else begin
  ClearAddIns(SP);
end;
ClearAddIns(SPml);
if (CStack[SP].SType in [ConstOnly,ConstTimesReg]) and
   (CStack[SP].ConstVal = 0.0) then FlagError(300);
if (CStack[SPml].SType in [ConstOnly,ConstTimesReg]) and
   (CStack[SPml].ConstVal = 0.0) then begin
   CStack[SPml].SType := ConstOnly;
   CStack[SPml].IsReal := true;
end else begin
   if not CStack[SP].IsReal then i2fpTop;
   if not CStack[SPml].IsReal then i2fpNext;
if CStack[SP].SType in [ConstOnly,ConstTimesReg] then
   case CStack[SPml].SType of
     ConstTimesReg,
     ConstOnly : CStack[SPml].ConstVal := CStack[SPml].ConstVal
                 / CStack[SP].ConstVal;
     RegOnly : begin
      CStack[SPml].ConstVal := 1.0
                 / CStack[SP].ConstVal;
      CStack[SPml].SType := ConstTimesReg;
     end;
     ConstPlusReg,
     Empty : FlagError(-515);
   end;
if CStack[SP].SType in [RegOnly,ConstTimesReg] then
   case CStack[SPml].SType of
     ConstTimesReg,
     RegOnly : CListOutCfdiv.x fp'+
               TopStr+,fp'+NextStr,2,$f200,
               $0020 + $400*nTop + $80*nNext,0,0);
     ConstOnly : begin
      CListOut('fmove.l # l,fp '+NextStr,4,
                 $f23c,$4000 + $80*nNext,$0000,$0001);
      CListOut('fdiv.x fp'+
                 TopStr+,fp'+NextStr,2,$f200,
                 $0020 + $400*nTop + $80*nNext,0,0);
      CStack[SPml].SType := ConstTimesReg;
     end;
     ConstPlusReg,
     Empty : FlagError(-516);
   end;
end;
DecSP;
end;
IntDiv : begin
ExpandCC(SP);
if CStack[SP].BaseAddr<>NoBase then FlagError(-517);
if CStack[SPml].BaseAddr<>NoBase then FlagError(-518);
if CStack[SP].SType = ConstOnly then begin
   if CStack[SPml].SType = ConstOnly then begin
     // Further code
   end else begin
     // Further code
   end;
end;

CStack[SPm1].ConstVal := GBTrunc(CStack[SPm1].ConstVal); 
DIV GBTrunc(CStack[SP].ConstVal);
decSP;
end else begin
CVal := GBTrunc(CStack[SP].ConstVal);
str(CVal,CStr);
MultInConstant(SPm1);
AddInConstant(SPm1);
ShiftVal := 0;
while (CVal and 1) = 0 do begin
CVal := CVal shr 1;
inc(ShiftVal);
end;
if CVal = 1 then begin
if ShiftVal <> 0 then begin
DoShLabl := NewLabel;
DoneLabl := NewLabel;
CListOut('tst.ld'+NextStr,1,$4a80+nNext,0,0,0);
DoBranch(CondGE,DoShLabl);
CVal := -1;
for i := 1 to ShiftVal do CVal := 2 * CVal;
str(CVal,ShiftStr);
CCmp(L,'# '+ShiftStr,'d'+NextStr,1,
    $e080+$200*(ShiftVal mod 8)+nNext,0,0,0);
end else begin
DoMoveQ(nTop,ShiftVal);
CListOut('asrl #'+ShiftStr+'d'+NextStr,1,
    $e080+$200*nTop+nNext,0,0,0);
end;
end;
CVal := GBTrunc(CStack[SP].ConstVal);
str(CVal,CStr);
CListOut('divs.ld #'+CStr+'d'+NextStr+'d'+TopStr,4,
    $4c7c,$0800+$1000*nNext,Hi Word(CVal),
    LoWord(CVal));
end;
decSP;
end;
end else begin
MultInConstant(SP);
AddInConstant(SP);
MultInConstant(SPm1);
AddlnConstant(SPml);
CLListOut('divsl d'+TopStr+',d'+NextStr,2,
$4c40 + nTop,$0800 + $1000*nNext + nNext,0,0);
DecSP;
end;
end;

Modulo : begin
ExpandCC(SP);
if CStack[SP].BaseAddr<>NoBase then FlagError(-519);
if CStack[SPml].BaseAddr<>NoBase then FlagError(-520);
if CStack[SP].STYPE = ConstOnly then begin
if CStack[SPml].STYPE = ConstOnly then begin
CStack[SPml].ConstVal := GBTrunc(CStack[SPml].ConstVal)
MOD GBTrunc(CStack[SP].ConstVal);
decSP;
end else begin
CVal := GBTrunc(CStack[SP].ConstVal);
str(CVal,CStr);
MultInConstant(SPml);
AddInConstant(SPml);
CLListOut('divsl #,d'+CStr+',d'+NextStr+',d'+TopStr,4,
$4c7c,$0800+$1000*nNext + nTop,HiWord(CVal),
LoWord(CVal));
CMove(L,'d'+TopStr,'d'+NextStr);
decSP;
end;
end else begin
MultInConstant(SP);
AddInConstant(SP);
MultInConstant(SPml);
AddInConstant(SPml);
CLListOut('divsl d'+TopStr+',d'+NextStr,2,
$4c40 + nTop,$0800+$1000*nNext + nTop,0,0);
CMove(L,'d'+TopStr,'d'+NextStr);
decSP;
end;
end;

IntConstant: begin
incSP;
with CStack[SP] do begin
STYPE := ConstOnly;
ConstVal := Param;
end;
end;

BoolConstant : begin
incSP;
with CStack[SP] do begin
STYPE := CondC;
CondField := 1 - Param;
end;
end;
RealConstant: begin
  GetRealConstant(RConst,Param);
  incSP;
  with CStack[SP] do begin
    SType := ConstOnly;
    IsReal := true;
    ConstVal := RConst;
  end;
end;

AddConstant: begin
  with CStack[SP] do begin
    case SType of
      Empty: begin
        SType := ConstOnly;
        ConstVal := Param;
      end;
      ConstOnly: ConstVal := ConstVal + Param;
      RegOnly: begin
        SType := ConstPlusReg;
        ConstVal := Param;
      end;
      ConstPlusReg: ConstVal := ConstVal + Param;
      ConstTimesReg: begin
        MultlnConstant(SP);
        ConstVal := Param;
        SType := ConstPlusReg;
      end;
    end;
  end;
end;

MultConstant: begin
  if CStack[SP].BaseAddr<>NoBase then FlagError(-521);
  ClearAddIns(SP);
  case CStack[SP].SType of
    ConstOnly, ConstTimesReg: CStack[SP].ConstVal :=
    CStack[SP].ConstVal * Param;
    RegOnly: begin
      CStack[SP].ConstVal := Param;
      CStack[SP].SType := ConstTimesReg;
    end;
    Empty, ConstPlusReg: FlagError(-522);
  end;
end;

ChkRange: begin
  ExpandCC(SP);
  {Param points to type table}
  if (TypeTabl^[Param].TypeDesc in [IntType.ByteType,
    ShortintType,WordType,EnumType,SubrangeType]) AND
  CheckOption('R') then begin
ClearAddIns(SP);
if CStack[SP].SType=ConstOnly then begin
  if TypeTabl'[Param].TypeDesc = EnumType then begin
    if (CStack[SP].ConstVal < 0) or (CStack[SP].ConstVal
        >= TypeTabl'[Param].p2) then FlagError(214);
  end else begin
    if (CStack[SP].ConstVal < TypeTabl'[Param].p1)
        or (CStack[SP].ConstVal > TypeTabl'[Param].p2) then
        FlagError(214);
  end;
end else begin
  AddInConstant(SP);
  if TypeTabl'[Param].TypeDesc = EnumType then begin
    CListOut('tst.l d' + TopStr,1,$4a80 + nTop,0,0,0);
    CListOut('traplt',1,$5dfc,0,0,0);
    str(TypeTabl'[Param].p2,CStr);
    CComp(L,'#'+Cstr,'d'+TopStr);
    CListOut('trapge1',1,$5ecf,0,0,0);
  end else begin
    str(TypeTabl'[Param].p1,CStr);
    CComp(L,'#'+Cstr,'d'+TopStr);
    CListOut('traplt',1,$5dfc,0,0,0);
    str(TypeTabl'[Param].p2,CStr);
    CComp(L,'#'+Cstr,'d'+TopStr);
    CListOut('trapgt1',1,$5ecf,0,0,0);
  end;
end;
end;
DoAbs : begin
  ExpandCC(SP);
  if CStack[SP].BaseAddr<>NoBase then FlagError(-523);
  if not CStack[SP].IsReal then FlagError(-524);
  MultInConstant(SP);
  AddInConstant(SP);
  CListOut('fabs.x fp'+TopStr,
            2,$f200,$0018 + $0480 * nTop,0,0);
  end;
DoSqrt,
DoSqr,
DoRound,
DoSin,
DoCos,
DoTan,
DoASin,
DoACos,
DoATan,
DoExp,
DoLn : begin
  ExpandCC(SP);
  if CStack[SP].BaseAddr<>NoBase then FlagError(-523);
if not CStack[SP].IsReal then i2fpTop;
MultlnConstant(SP);
AddlnConstant(SP);
case instr of
  DoSqrt: CListOut('fsqrt.x fp'+TopStr,
    2,$f200,$0004 + $0480 * nTop,0,0);
  DoSqr: CListOut('fml.x fp'+TopStr+',fp'+TopStr,
    2,$f200,$0023 + $400*ntop + $80*nTop,0,0);
  DoSin: CListOut('fnsin.x fp'+TopStr,
    2,$f200,$000e + $0480 * nTop,0,0);
  DoCos: CListOut('fcos.x fp'+TopStr,
    2,$f200,$001d + $0480 * nTop,0,0);
  DoTan: CListOut('ftan.x fp'+TopStr,
    2,$f200,$000f + $0480 * nTop,0,0);
  DoASin: CListOut('fasin.x fp'+TopStr,
    2,$f200,$000c + $0480 * nTop,0,0);
  DoACos: CListOut('facos.x fp'+TopStr,
    2,$f200,$001c + $0480 * nTop,0,0);
  DoATan: CListOut('fatan.x fp'+TopStr,
    2,$f200,$000a + $0480 * nTop,0,0);
  DoExp: CListOut('fexp.x fp'+TopStr,
    2,$f200,$0010 + $0480 * nTop,0,0);
  DoLn: CListOut('flogn.x fp'+TopStr,
    2,$f200,$0014 + $0480 * nTop,0,0);
  DoRound: begin
    CListOut('fmove.l fp' + TopStr + ',d'  + TopStr,
      2, $f200 + nTop, $6000 + $80 * nTop, 0, 0);
    CStack[SP].IsReal := False;
    end;
end;
end;
end.
unit emitfor;
interface
uses gencode,types,tables,errors,list,strstore,options,labels,CircRef;
procedure EmitForCase(instr:OpCode;Param:longint);
implementation
procedure EmitForCase(instr:OpCode;Param:longint);
var
  i : integer;
  RConst : double;
  RSize : longint;
  CVal : longint;
  CStr : string[25];
begin
case instr of
  EvalCase : begin
    ExpandCC(SP);
    MultlnConstant(SP);
    AddBaseAddr(SP);
    AddInConstant(SP);
    if SPV <> 0 then FlagError(-526);
    if SP <> 0 then CMove(L,'d'+TopStr,'d0');
    decSP;
  end;
  CaseCmp : begin
    if CStack[SP].SType <> ConstOnly then FlagError(-527);
    if CStack[SP].BaseAddr <> NoBase then FlagError(-528);
    if CStack[SP].IsReal then FlagError(-529);
    CVal := GBTrunc(CStack[SP].ConstVal);
    str(CVal.CStr);
    CCmp(L,'# '+ CStr/dO');
    if Param > 0 then DoBranchfCondEQ,Param)
      else DoBranchfCondNE,-Param);
    DecSP;
  end;
  ForInc : begin
    if SPV <> -1 then FlagError(-530);
    TDesc := TypeTabl^[Param].TypeDesc;
    if TDesc=SubrangeType then
      TDesc := TypeTabl'[TypeTabl^[Param].SubType].
      TypeDesc;
    if ParEnabled then CMove(L,'8(a7)','a0')
      else CMove(L,'4(a7)','a0');
    case TDesc of
      ByteType,ShortintType,
      EnumType,CharType,
      BoolType : CAdd(B,'#1','(a0)');
      WordType,
      IntType : CAdd(W,'#1','(a0)');
      LongintType : CAdd(L,'#1','(a0)');
      else FlagError(-531);
    end;
    if ParEnabled then CAdd(L,'#1',4(a7))
      else CAdd(L,'#1','a7');
    incSP;
    CStack[SP].SType := CondC;
    CStack[SP].CondField := CondLE;
  end;
  ForDec : begin
    if SPV <> -1 then FlagError(-532);
    TDesc := TypeTabl^[Param].TypeDesc;
    if TDesc=SubrangeType then
      TDesc := TypeTabl'[TypeTabl^[Param].SubType].
      TypeDesc;
    if ParEnabled then CMove(L,'8(a7)','a0')
else CMove(L,'4(a7)',a0');
case TDesc of
  ByteType, ShortintType, EnumType, CharType, BoolType:
    CSub(B,'#1','a0');
  WordType, IntType:
    CSub(W,'#1','a0');
  LongintType:
    CSub(L,'#1','a0');
else FlagError(-533);
end;
if ParEnabled then CSub(L,'#r','4(a7)')
else CSub(L,'#17(a7)')
incSP;
CStack[SP].SType := CondC;
CStack[SP].CondField := CondGE;
end;

ForPrelim: begin
  {three values are on the stack}
  {TOP ==> final value }
  {2ND ==> initial value }
  {BOT ==> var address }
  {Param points into the type table}
  if ParEnabled then begin
    if SP = 7 then FlagError(-777);
    CMove(L,'(a7) +','d7');
  end;
  ExpandCC(SPm1);
  MultlnConstant(SPm1);
  AddBaseAddr(SPm1);
  ClearAddIns(SPm1);
  GetFullAddr(SPm2);
  {Param points into the type table at entry for variable}
  TDesc := TypeTable^[Param].TypeDesc;
  if TDesc=SubrangeType then
    TDesc := TypeTable^[TypeTable^[Param].SubType].TypeDesc;
  if CStack[SPm1].IsReal or CStack[SP].IsReal then
    FlagError(-534);
  case TDesc of
    IntType,
    WordType: if CStack[SPm1].SType=ConstOnly then begin
      CVal := GBTrunc(CStack[SPm1].ConstVal);
      str(CVal,CStr);
      CMove(W,'#'+CStr,AddrStr);
    end else CMove(W,'d'+NextStr,+ AddrStr);
end else CMove(B,'d'+NextStr,+ AddrStr);
LongIntType : if CStack[SPm1].SType=ConstOnly then begin
  CVal := GBTrunc(CStack[SPm1].ConstVal);
  str(CVal,CStr);
  CMove(L,'# '+CStr,AddrStr);
end else CMove(L,'d'+NextStr,+ AddrStr);
else FlagError(6273);
end;
PushEA;
ExpandCC(SP);
MultiNConstant(SP);
MultiNConstant(SPm1);
SetTypes;
if CStack[SP].SType in [ConstOnly,ConstPlusReg] then begin
  if CStack[SPm1].SType in [ConstOnly,ConstPlusReg] then
  begin
    CStack[SP].ConstVal := CStack[SP].ConstVal -
    CStack[SPm1].ConstVal
  end else begin
    CStack[SP].ConstVal := -CStack[SP].ConstVal;
    case CStack[SPm1].SType of
      Empty : CStack[SP].SType := ConstOnly;
      RegOnly : CStack[SP].SType := ConstPlusReg;
    end;
  end;
if CStack[SP].SType in [RegOnly,ConstPlusReg] then begin
  if CStack[SPm1].SType in [RegOnly,ConstPlusReg] then
  begin
    CSub(L,'d'+TopStr,'d'+NextStr)
  end else begin
    CStack[SP].BaseAddr := CStack[SPm1].BaseAddr;
    CStack[SP].BLevel := CStack[SPm1].BLevel;
    CStack[SP].ConstVal := CStack[SPm1].ConstVal;
    case CStack[SPm1].SType of
      Empty : CStack[SP].SType := RegOnly;
      ConstOnly : CStack[SP].SType := ConstPlusReg;
    end;
    CStack[SPm1].SType := NewDead;
    CListOut(Cneg,l 'd'+TopStr,l,
    $4480 + nTop,0,0,0);
  end;
end;
decSP;
ClearAddIns(SP);
if CStack[SP].SType = ConstOnly then begin
  CVal := GBTrunc(CStack[SP].ConstVal);
  str(CVal,CStr);
  CMove(L,'#'+CStr,'-(a7)');
end
else CMove(L,'d'+TopStr,'-(a7)');
decSP;
decSP;
if ParEnabled then CMove(L,'d7','(a7)');
AdjustStack(8);
end;
ToCmp, DowntoCmp : begin
if SPV o - 1 then FlagError(-420);
{Param is the label number of the loop end}
if ParEnabled then
  CListOut('tst.l 4(a7),2,$4aaf,$0004,0,0)
else
  CListOut('tst.l (a7),1,$4a97,0,0,0);
if instr = ToCmp then DoBranch(CondGT,Param)
else DoBranch(CondLT,Param);
end;
ForEnd : begin
  if ParEnabled then CMove(L,'(a7)',8(a7));
  CAdd(L,'# 8','a7');
  AdjustStack(-8);
end;
EnablePar : ParEnabled := true;
DisablePar : ParEnabled := false;
else FlagError(-535);
end;
end.
unit emitcnt;
interface
uses gencode,types,tables,errors,list,strstore,options,labels,lblock,CircRef;
procedure EmitControl(instr:OpCode;Param:longint);
implementation
var
  DoTrace : boolean;
procedure EmitControl(instr:OpCode;Param:longint);
var LablName : string[25];
  MemSize : longint;
  MStr : string[6];
  i : integer;
  RConst : double;
  L1,L2,
  RSize : longint;
  CVal : longint;
  CStr : string[25];
  StoreOffset : boolean;
begin
  case instr of
    DecLabel : PlaceLabel(Param);
JmpTrue : begin
    StoreOffset := true;
    str(Param, LablName);
    if CStack[SP].SType = CondC then begin
        case CStack[SP].CondField of
            CondCC : CListOut('bcc.l L' + LablName, 3, $64ff, 0, 0, 0);
            CondCS : CListOut('bcs.l L' + LablName, 3, $65ff, 0, 0, 0);
            CondEQ : CListOut('beq.l L' + LablName, 3, $67ff, 0, 0, 0);
            CondGE : CListOut('bge.l L' + LablName, 3, $6cff, 0, 0, 0);
            CondGT : CListOut('bgt.l L' + LablName, 3, $6eff, 0, 0, 0);
            CondHI : CListOut('bhi.l L' + LablName, 3, $62ff, 0, 0, 0);
            CondLE : CListOut('ble.l L' + LablName, 3, $6fff, 0, 0, 0);
            CondLS : CListOut('blsa.l L' + LablName, 3, $63ff, 0, 0, 0);
            CondLT : CListOut('blt.l L' + LablName, 3, $6dff, 0, 0, 0);
            CondMI : CListOut('bmi.l L' + LablName, 3, $6bff, 0, 0, 0);
            CondNE : CListOut('bne.l L' + LablName, 3, $66ff, 0, 0, 0);
            CondPL : CListOut('bpl.l L' + LablName, 3, $6aff, 0, 0, 0);
            CondVC : CListOut('bvc.l L' + LablName, 3, $68ff, 0, 0, 0);
            CondVS : CListOut('bvs.l L' + LablName, 3, $69ff, 0, 0, 0);
            CondT : CListOut('bra.l L' + LablName, 3, $60ff, 0, 0, 0);
            CondF : StoreOffset := false;
        end;
        CStack[SP].SType := Empty; {prevent error}
    end else begin
        ExpandCC(SP);
        MultInConstant(SP);
        AddBaseAddr(SP);
        AddInConstant(SP);
        CListOut('btsl. #0d' + TopStr, 2, $0800 + nTop, $0000, 0, 0);
        CListOut('bne.l L' + LablName, 3, $66ff, $0000, $0000, 0);
    end;
    if StoreOffset then StoreLabelOffs(Param);
    DecSP;
end;
end;
JmpFalse : begin
    StoreOffset := true;
    str(Param, LablName);
    if CStack[SP].SType = CondC then begin
        case CStack[SP].CondField of
            CondCC : CListOut('bcc.l L' + LablName, 3, $64ff, 0, 0, 0);
            CondCS : CListOut('bcs.l L' + LablName, 3, $65ff, 0, 0, 0);
            CondEQ : CListOut('beq.l L' + LablName, 3, $67ff, 0, 0, 0);
            CondGE : CListOut('bge.l L' + LablName, 3, $6cff, 0, 0, 0);
            CondGT : CListOut('bgt.l L' + LablName, 3, $6eff, 0, 0, 0);
            CondHI : CListOut('bhi.l L' + LablName, 3, $62ff, 0, 0, 0);
            CondLE : CListOut('ble.l L' + LablName, 3, $6fff, 0, 0, 0);
            CondLS : CListOut('blsa.l L' + LablName, 3, $63ff, 0, 0, 0);
            CondLT : CListOut('blt.l L' + LablName, 3, $6dff, 0, 0, 0);
            CondMI : CListOut('bmi.l L' + LablName, 3, $6bff, 0, 0, 0);
            CondNE : CListOut('bne.l L' + LablName, 3, $66ff, 0, 0, 0);
            CondPL : CListOut('bpl.l L' + LablName, 3, $6aff, 0, 0, 0);
            CondVC : CListOut('bvc.l L' + LablName, 3, $68ff, 0, 0, 0);
            CondVS : CListOut('bvs.l L' + LablName, 3, $69ff, 0, 0, 0);
            CondT : CListOut('bra.l L' + LablName, 3, $60ff, 0, 0, 0);
            CondF : StoreOffset := false;
        end;
        CStack[SP].SType := Empty; {prevent error}
    end else begin
        ExpandCC(SP);
        MultInConstant(SP);
        AddBaseAddr(SP);
        AddInConstant(SP);
        CListOut('btsl. #0d' + TopStr, 2, $0800 + nTop, $0000, 0, 0);
        CListOut('bne.l L' + LablName, 3, $66ff, $0000, $0000, 0);
    end;
    if StoreOffset then StoreLabelOffs(Param);
    DecSP;
end;
CondVC :  CListOut('bvs.l L '+LablName,3,$69ff,0,0,0);
CondVS :  CListOut('bvc.l L '+LablName,3,$68ff,0,0,0);
CondF :  CListOut('bra.l L '+LablName,3,$60ff,0,0,0);
CondT :  StoreOffset := false;
end;
CStack[SP].SType := Empty; {prevent error}
end else begin
 ExpandCC(SP);
 MultInConstant(SP);
 AddBaseAddr(SP);
 AddlnConstant(SP);
 CListOut('btst.l #0,d' +TopStr,2,$0800+nTop,$0000,0,0);
 CListOut('beq.l L'+LablName,3,$67ff,$0000,$0000,0);
end;
if StoreOffset then StoreLabelOffs(Param);
DecSP;
end;
JmpAlways :  begin
 if Param=0 then begin
  CListOut('section 9’,0,0,0,0,0);
  if RomCode and CheckOption(‘K’) then begin
   CListOut('dc.l $' + HexStrL(RomVarBase - $1000,8)
   + ' * Initial ISR’,
   HiWord(RomVarBase - $1000),
   LoWord(RomVarBase - $1000),0,0);
   CListOut('dc.l $' + HexStrL(RomCodBase+8,8)
   + ' * Initial PC,2,
   HiWord(RomCodBase+8),
   LoWord(RomCodBase+8),0,0);
   end;
  end;
  str(Param,LablName);
  CListOut('bra.l L '+LablName,3,$60ff,$0000,$0000,0);
  StoreLabelOffs(Param);
  end;
ProStart : begin {Param contains global variable size}
 A3Valid := false;
 MemSize := DefaultStack + VarAddrs[BlockLevel];
 str(MemSize,MStr);
 CListOut('dc.l '+MStr+' *Variable + Stack Size’,2,
 HiWord(MemSize),LoWord(MemSize),0,0);
 PlaceLabel(O);
 if RomCode and CheckOption(‘K’) then begin
  CMove(L,’#$ '+HexStrL(RomVarBase-$2000,8),’d0’);
  CListOut('movecl d0,vbr’,2,$4e7b,$0801,0,0);
  CListOut('andi #$efff,SR’,2,$027c,$efff,0,0);
  CMove(L,’#$ '+HexStrL(RomVarBase - $2000 + $400,8),’a7’);
  CListOut('ori #$1700,SR’,2,$007c,$1700,0,0);
  CMove(L,’#$ '+HexStrL(RomVarBase - $1000,8),’a7’);
  end;
  if (Param <> 0) and (not RomCode) then begin

str(Param,NumStr);
CSub(L,'#'+NumStr,'a7');
end;
if not RomCode then CMove(L,'a7','a6');
end;

ProgEnd : OSFunc(0);
SourceCodeEnd : CListOut('end',0,0,0,0,0);
ExProcStart : begin
  A3Valid := false;
  if CheckOption('Z') then begin
    {if this is a throwaway stack frame, get off the int stack.}
    L1 := NewLabel; L2 := NewLabel;
    CMove(L,'d0',-(a7));
    CMove(W,'10(a7)','d0');
    CListOut('andi.w #$f000,d0',2,$0240,$f000,0,0);
    CCmp(W,'# $1000','d0');
    DoBranch(CondNE,L1);
    CMove(L,'(a7)+','d0');
    CAdd(L,'#8','a7');
    CListOut('ori.w #$1000,sp,2,007c,$1000,0,0,0); DoBranch(CondT,L2);
    PlaceForgetLabel(L1);
    CMove(L,'a7','d0'); PlaceForgetLabel(L2);
  end;
  CListOut('movem.l d0-d7/a0-a7,-(a7)';2,$48e7,$ffff,0,0,0);
  CMove(L,'a7','a4');
  CListOut('move USP,a0',1,$4e68,0,0,0);
  CMove(L,'a0','60(a7)');
  if Param <> 0 then begin
    str(Param,NumStr);
    CSub(L,'#'+NumStr,'a7');
  end;
  CMove(L,'a5','(a7)');
  CMove(L,'a7','a5');
end;
ExProcEnd : begin
  str(Param,NumStr);
  CMove(L,'(a7)+','a5');
  if Param <> 0 then
    CAdd(L,'#'+NumStr,'a7');
  CMove(L,'60(a7)','a0');
  CListOut('move a0,USP',1,$4e68,0,0,0,0);
  CListOut('movem.l (a7)+,d0-d7/a0-a6',2,$4cdf,$7fff,0,0,0);
  CAdd(L,'#4','a7');
  CListOut('rte',1,$4e73,0,0,0,0);
end;
ProcStart,
FuncStart : begin
  A3Valid := false;
  {Param contains local variable size}

if Param <> 0 then begin
  str(Param,NumStr);
  CSub(L,'#'+NumStr,'a7');
end;
CMove(L,'a5','-(a7)');
CMove(L,'a7','a5');
end;

ProcEnd,

FuncEnd : begin
  {Param contains local variable size}
  str(Param,NumStr);
  CMove(L,'(a7)+','a5');
  if Param <> 0 then
    CAdd(L,'#'+NumStr,'a7');
  CListOutCrts',l,$4e75,0,0,0);
end;

PrepFuncCall : begin
  {Param is pointer into FuncTabl}
  {This routine saves room on the stack for the func result}
  RSize := TypeTabl^ [FuncTabl|^ Param].TypePointer.
    TypeSize;
  RSize := (RSize + 1) and $fffffffe;
  str(RSize,NumStr);
  CSub(L,'# '+NumStr,'a7');
end;

FuncCall : begin
  {Param is pointer into FuncTabl}
  if (FuncTabl^ [Param].level=BlockLevel) or
    (FuncTabl^ [Param].level=0) then begin
    CMove(L,'a4','a3');
    CMove(L,'a7','a4');
    CMove(L,'a3','-(a7)');
    A3Valid := false;
  end else begin
    incSP;
    CStack[SP].SType := Empty;
    CStack[SP].ConstVal := 0;
    CStack[SP].BaseAddr := ParamAddr;
    CStack[SP].BLevel := FuncTabl^ [Param].level;
    GetFullAddr(SP);
    PushEA;
    decSP;
    CMove(L,'a4','-(a7)');
    CLea('8(a7)','a4');
    incSP;
    CStack[SP].SType := Empty;
    CStack[SP].BaseAddr := VarAddr;
    CStack[SP].BLevel := FuncTabl^ [Param].level;
    GetFullAddr(SP);
    CMove(L,'a5','-(a7)');
    CLea(AddrStr,'a5');
decSP;
end;
SaveStk;
str(FuncTabl^ [Param].CodeLabel,NumStr);
CListOut('bsr L'+NumStr,3,$61ff,0,0,0);
StoreLabelOffs(FuncTabl^ [Param].CodeLabel);
RestoreStk;
A3Valid := false;
if (FuncTabl^ [Param].level=BlockLevel) or
  (FuncTabl'' [Param].level=0) then
  CMove(L,'(a7)++',a4')
else begin
  CMove(L,'(a7)++',a5');
  CMove(L,'(a7)++',a4');
  CAdd(L,'#4',a7');
end;
{remove Parameters from the stack}
if FuncTabl' [Param].RsltOffset <> 0 then begin
  str(FuncTabl' [Param].RsltOffset,NumStr);
  CAdd (L,'# '+ NumStr,a7') ;
end;
TDesc := TypeTabl' [FuncTabl' [Param].TypePointer].TypeDesc;
if TDesc=SubrangeType then
  TDesc :=TypeTabl' [TypeTabl' [FuncTabl' [Param].TypePointer].
              SubType].TypeDesc;
incSP;
CStack[SP].SType := RegOnly;
if TDesc in [ByteType,WordType,CharType,BoolType,EnumType]
  then CClr(L,'d'+TopStr);
case TDesc of
  LongintType,
    PtnType : CMove(L,'(a7)++',d'+TopStr);
  IntType,
    WordType : CMove(W,'(a7)++',d'+TopStr);
  ShortintType,
    ByteType,
    CharType,
    BoolType,
    EnumType : begin
    CMove(B,'(a7)',d'+TopStr);
    CAdd(L,'#2',a7');
    end;
  RealType : begin
    CListOut('fmove.s (a7)+,fp'+TopStr,2,$f21f, $4400+nTop*$80,0,0);
    CStack[SP].IsReal := true;
  end;
end;
DoubleType : begin
  CListOut('fmove.d (a7)+,fp'+TopStr,2,$f21f, $5400+nTop*$80,0,0);
  CStack[SP].IsReal := true;
end;
StringType : CStack[SP].STYPE := StrOnly;
else FlagError(5273);
end;
if TDesc=IntType
then CListOut('ext.l d'+TopStr,l,$48cO+nTop,0,0,0);
if TDesc=ShortintType
then CListOut('extbl. d'+TopStr,l,$49c0+nTop,0,0,0);
end;
ProcCall : begin
{Param is pointer into ProcTabl}
if (ProcTabl^[Param].level=BlockLevel) or
(ProcTabl^[Param].level=0) then begin
CMove(L,'a4','a3');
CMove(L,'a7','a4');
CMove(L,'a3',-(a7));
A3Valid := false;
end else begin
incSP;
CStack[SP].STYPE := Empty;
CStack[SP].ConstVal := 0;
CStack[SP].BaseAddr := ParamAddr;
CStack[SP].BLevel := ProcTabl^[Param].level;
GetFullAddr(SP);
PushEA;
decSP;
CMove(L,'a4',-(a7));
CLea('8(a7)',a4');
incSP;
CStack[SP].STYPE := ConstOnly;
CStack[SP].ConstVal := -4;
CStack[SP].BaseAddr := VarAddr;
CStack[SP].BLevel := ProcTabl^[Param].level;
GetFullAddr(SP);
CMove(L,'a5',-(a7));
CLea(AddrStr,a5');
decSP;
end;
SaveStk;
str(ProcTabl^[Param].CodeLabel,NumStr);
CListOut('bsr.l L'+ NumStr,3,$61ff,0,0,0);
StoreLabelOffs(ProcTabl^[Param].CodeLabel);
RestoreStk;
A3Valid := false;
if (ProcTabl^[Param].level=BlockLevel) or
(ProcTabl^[Param].level=0) then
CMove(L,`(a7)`+`,a4`)
else begin
CMove(L,`(a7)`+`,a5`);
CMove(L,`(a7)`+`,a4`);
CAdd(L,`#4`,a7);
}
end;

{remove Parameters from the stack}
if ProcTabl^[Param].ParamSize <> 0 then begin
    str(ProcTabl^[Param].ParamSize,NumStr);
    CAdd(L:'#'+NumStr,'a7');
end;
end;

Trace : begin
    if DoTrace or CheckOption('T')
    then CListOut(' trap # 0 ',l,$4e40,0,0,0);
end;
else FlagError(-32);
end;

procedure CheckForTrace;
var
    i : byte;
begin
    DoTrace := false;
    for i := 2 to ParamCount do if ParamStr(i) = '/t' then
        DoTrace := true;
end;

begin
    CheckForTrace;
end.

unit lblock;
{used to keep track of current block level}

interface
uses errors,tables/types;
const MaxLevel = 16;

var VarAddrs : array[0..MaxLevel] of longint;
procedure StartNewBlock;
procedure EndBlock;

implementation
procedure StartNewBlock;
begin
    inc(BlockLevel);
    if BlockLevel>MaxLevel then begin
        FlagError(251);
        dec(BlockLevel);
    end;
    VarAddrs[BlockLevel] := 0;
end;

procedure EndBlock;
begin
while (IdTablPtr<>0) AND (IdTabl^[IdTablPtr].level=BlockLevel) do begin
  with IdTabl^[IdTablPtr] do
  begin
    if IdTyp = FwdProcName then
      with ProcTabl^[valptr] do
        FlagErrorPos(117,row,col,fhum);
    if IdTyp = DecRsrcProc then
      with FieldTabl^[valptr] do
        FlagErrorPos(291,row,col,fhum);
    if IdTyp = FwdFuncName then
      with FuncTabl^[valptr] do
        FlagErrorPos(117,row,col,fhum);
    if (IdTyp=TypeName) and (TypeTabl^[valptr].TypeDesc=FwdPtrType)
      then FlagErrorPos(117,TypeTabl^[valptr].SubType,
        TypeTabl^[valptr].p1,TypeTabl^[valptr].p2);
  end;
  dec(IdTablPtr);
end;
while (TypeTablPtr<>0) AND (TypeTabl^[TypeTablPtr].level=BlockLevel) do begin
  if TypeTabl^[TypeTablPtr].TypeDesc=EnumType then
    EnumTablPtr := TypeTabl^[TypeTablPtr].p1;
  if TypeTabl^[TypeTablPtr].TypeDesc=RecordType then
    FieldTablPtr := TypeTabl^[TypeTablPtr].p1;
  if TypeTabl^[TypeTablPtr].TypeDesc=ProcessType then
    ParamTablPtr := TypeTabl^[TypeTablPtr].p1 - 1;
  if TypeTabl^[TypeTablPtr].TypeDesc=ResourceType then
    FieldTablPtr := TypeTabl^[TypeTablPtr].p1;
  dec(TypeTablPtr);
end;
while (VarTablPtr<>0) AND (VarTabl^[VarTablPtr].level=BlockLevel) do dec(VarTablPtr);
while (ProcTablPtr<>0) AND (ProcTabl^[ProcTablPtr].level=BlockLevel) do begin
  if ParamTablPtr >= ProcTabl^[ProcTablPtr].FirstParam then
    ParamTablPtr := ProcTabl^[ProcTablPtr].FirstParam-1;
  dec(ProcTablPtr);
end;
while (FuncTablPtr<>0) AND (FuncTabl^[FuncTablPtr].level=BlockLevel) do begin
  if ParamTablPtr >= FuncTabl^[FuncTablPtr].FirstParam then
    ParamTablPtr := FuncTabl^[FuncTablPtr].FirstParam-1;
  dec(FuncTablPtr);
end;
  dec(BlockLevel);
end;
end.
unit emitinit;
interface
uses genCode, types, tables, errors, list, strstore, options, labels, CircRef;
procedure EmitInitialization(instr: OpCode; Param: longint);

implementation

procedure EmitInitialization(instr: OpCode; Param: longint);
var
  i : integer;
  RConst : double;
  RSize : longint;
  CVal : longint;
  CStr : string[10];
  DoInit : boolean;

function RequiresInit(TPtr: integer): boolean;
begin
  RequiresInit := ContainsType(FileType, TPtr, 0) or
                 ContainsType(TextType, TPtr, 0) or
                 ContainsType(ChannelType, TPtr, 0) or
                 (ContainsType(ProcessType, TPtr, 0) AND DoInit) or
                 ContainsType(ResourceType, TPtr, 0) or
                 ContainsType(SharedType, TPtr, 0) or
                 ContainsType(TemplateType, TPtr, 0);
end;

procedure InitFile(TPtr: integer);
var LI : longint; LStr : string[10];
begin
  CMove(L, ' (a7) ', 'a0' );
  if DoInit then CMove(B, '# 0 ', 'd0' )
  else begin
    LI := NewLabel;
    CMove(B, ' (a0) ', 'd0' );
    CCmp(B, '# 1 ', 'd0' );
    DoBranch(CondLT, LI);
    CCmp(B, '# 2 ', 'd0' );
    DoBranch(CondGT, LI);
    if CheckOption('T') then DoMoveQ(2,1) else DoMoveQ(2,0);
    OSFunc($c);
    PlaceForgetLabel(L1);
  end;
end;

procedure InitChannel(TPtr: integer);
var
  CVal : longint;
  CStr : string[20];
begin
  if DoInit then begin
    CVal := TypeTabl^ [TypeTabl^ [TPtr].SubType].TypeSize;
    str(CVal, CStr);
  end;
end;
procedure InitProcess(TPtr:integer);
begin
  if DoInit then begin
    CMove(L,'(a7)','a0');
    CMove(W,'# 0 (a0)');
  end;
end;

procedure InitResource(TPtr:integer);
var DVal : longint;
DStr : string[10];
begin
  if DoInit then begin
    DVal := TypeTabl^[TPtr].SubType;  {stack size}
    str(DVal,DStr);
    CMove(L,'# '+DStr,'d1');
    DVal := TypeTabl^[TPtr].VariableSpace;  {variable size}
    str(DVal,DStr);
    CMove(L,'# '+DStr,'d2');
    OSFunc($40);
    CMove(L,'(a7)','a0');
    CMove(L,'d0',(a0));
  end;
end;

procedure InitTemplate(TPtr:integer);
var L1,
    CVal : longint;
CStr : string[12];
begin
  if DoInit then begin
    {set status to released}
    CMove(L,'(a7)','a0');
    CMove(W,'# 0 (a0)');  { Initial status }
    OSFunc($55);  {get template id}
    CMove(W,'d0',(a0)');  { dimension count into template}
  end;
end;
OSFunc($51); {get cpu,process info}
CMove(L,'d00', (a0)');
end else begin
{is it released? if not, release it}
CMove(L,'(a7)',(a0)');
CLiStOut(tst.w (a0)',1,4a50,0,0,0);
L1 := NewLabel;
DoBranch(CondEQ,L1);
CMove(W,' # 07 (a0) ');
CVal := TypeTabl^ [TPtr].Pl; {number of dimensions}
DoMoveQ(1,CVal);
OSFunc($52);
PlaceForgetLabel(L1);
end;
end;

procedure InitShared(TPtr:integer);
var CVal : longint;
CStr : string[10];
TmpPtr : integer;
Limit,
: word;
begin
if DoInit then begin
TmpPtr := TypeTabl^ [TPtr].SubType;
Limit := ArrayDepth(TmpPtr);
for i := 1 to Limit do begin
CVal := TypeTabl^ [TypeTabl^ [TmpPtr].p1].p1; str(CVal,CStr);
CMove(L,'# '+CStr,'- (a7) ');
CVal := TypeTabl^ [TypeTabl^ [TypeTabl^ [TmpPtr].p1].p2; str(CVal,CStr);
CMove(L,'# '+CStr,'- (a7) ');
if i <> Limit then TmpPtr := TypeTabl^ [TmpPtr].p2;
end;
str(TypeTabl^ [TypeTabl^ [TypeTabl^ [TmpPtr].p2].TypeSize,CStr);
CMove(L,'# '+CStr,d1'); {element size}
str(Limit,CStr);
CMove(L,'# '+CStr,d2'); { # dimensions}
OSFunc($60);
str(Limit*8,CStr);
CAdd(L,'# '+CStr,'a7');
CMove(L,'(a7)','a0');
CMove(L,'d00', (a0)');
end else begin
CMove(L,'(a7)',(a0)');
CMove(L,'(a0)','d00');
OSFunc($61);
end;
end;

procedure InitVariable(TPtr:integer);
var L1,CVal:longint; LStr,CStr : string[10];
ij,k : longint;
begin
  case TypeTabl^[TPtr].TypeDesc of
    ArrayType : begin
      if RequiresInit(TypeTabl^[TPtr].P2) then begin
        CVal := TypeTabl^[TypeTabl^[TPtr].P1].P2 -
          TypeTabl^[TypeTabl^[TPtr].P1].P1 + 1;
        str(CVal,CStr);
        CMove(L,'#' + CStr,' -(a7) ');
        LI := NewLabel;
        PlaceLabel(L1);
        InitVariable(TypeTabl^[TypeTabl^[TPtr].P2]);
        CVal := TypeTabl^[TypeTabl^[TPtr].P2].TypeSize;
        str(CVal,CStr);
        CMove(L,'#' + CStr,'0 ');
        CAdd(L,'#d0(a7)' );
        CSub(L,'#1 4(a7)' );
        DoBranch(CondNE,L1);
        ForgetLabel(L1);
        CAdd(L,'#87(a7)' );
      end;
    end;
    RecordType : begin
      j := TypeTabl^[TPtr].P1; {first field}
      k := TypeTabl^[TPtr].P2; {# of fields}
      for i := j to j+k-1 do
        if RequiresInit(FieldTabl^[i].FTypePtr) then begin
          CMove(L,'(a7)','d0');
          if FieldTabl^[i].AddrOffs <> 0 then begin
            CVal := FieldTabl^[i].AddrOffs;
            str(CVal,CStr);
            CAdd(L,'#' + CStr,'d0');
          end;
          CMove(L,'d0','-(a7)');
          InitVariable(FieldTabl^[i].FTypePtr);
          CAdd(L,'#47(a7)' );
        end;
      end;
    end;
    FileType,
    TextType : InitFile(TPtr);
    ChannelType : InitChannel(TPtr);
    ProcessType : InitProcess(TPtr);
    ResourceType : InitResource(TPtr);
    TemplateType : InitTemplate(TPtr);
    SharedType : InitShared(TPtr);
  end;
end;

begin
  case instr of
InitPF,
ClosePF: begin;
  if SPV <> -1 then FlagError(-562);
  DInit := instr=InitPF;
  i := VarTablPtr;
  while (i > 0) AND (VarTabl^[i].level = BlockLevel) do begin
    if (VarTabl^[i].IsAbsolute = NotAbs) and
      RequiresInit(VarTabl^[i].VarTypePtr) then begin
      {push the base address onto the stack}
      incSP;
      with CStack[SP] do begin
        if not (RomCode and (BlockLevel=0)) then begin
          BaseAddr := VarAddr;
          ConstVal := 0;
        end else ConstVal := RomVarBase;
        BLevel := BlockLevel;
        SType := ConstOnly;
        ConstVal := ConstVal + VarTabl^[i].AddrOffset;
      end;
      MultlnConstant(SP);
      GetFullAddr(SP);
      PushEA;
      decSP;
      InitVariable(VarTabl^[i].VarTypePtr);
      CAdd(L,'#47a7' );
    end;
    dec(i);
  end;
else FlagError(-563);
end;
end.

unit GenCode;

interface
uses types, tables, errors, list, strstore, options, labels, CircRef;

const
  TAB=' ,';
  CondCC = 4;
  CondCS = 5;
  CondEQ = 7;
  CondGE = 12;
  CondGT = 14;
  CondHI = 2;
  CondLE = 15;
  CondLS = 3;
  CondLT = 13;
  CondMI = 11;
CondNE = 6;
CondPL = 10;
CondVC = 8;
CondVS = 9;
CondT = 0;
CondF = 1;

const
MaxSp = 7;

type
AddrDescType = (NoBase,VarAddr,ParamAddr);

StackDataType = record
  SType : (Empty,ConstOnly,RegOnly,ConstPlusReg,ConstTimesReg,StrOnly,
           CondC,NewDead,OldDead);
  ConstVal : double; {value of possible constant}
  IsReal : boolean;
  BaseAddr : AddrDescType;
  BLevel : byte;
  CondField : byte;
end; {StackDataType}

var
SP,SPm1,SPm2 : integer;
CStack : array[0..MaxSp] of StackDataType;
nTop,nNext : shortint;
TopStr,NextStr,NumStr : string[25];
TDesc : TypeDescType;
AddrStr : string[40];
EAMode : byte;
EAReg : byte;
NumExtWords : byte;
Ext1,Ext2,Ext3,Ext4 : word;
NextLabel2Use : longint;
ParEnabled : boolean;
A3Valid : boolean;
A3AddrType : AddrDescType;
A3BlockLevel : byte;

function LoWord(l:longint):word;
function HiWord(l:longint):word;
function SPV : integer;
procedure GencodeInit;
procedure incSP;
procedure decSP;
procedure Simplify;
procedure ExpandCC(entry:byte);
procedure negate;
procedure i2fTop;
procedure i2fpNext;
procedure SetTypes;
procedure SaveStk;
procedure RestoreStk;
procedure GetFullAddr(entry:byte);
procedure AddBaseAddr(entry:byte);
procedure AddInConstant(entry:byte);
procedure ClearAddIns(entry:byte);
procedure MultiInConstant(entry:byte);
procedure DoMoveQ(reg:byte;val:byte);
procedure Trap2;
procedure OSFunc(N:byte);
procedure PushEA;
procedure CMove(MSize : SizeType;Source,Dest:string);
procedure CLea(Source,Dest:string);
procedure CAdd(MSize : SizeType;Source,Dest:string);
procedure CSub(MSize : SizeType;Source,Dest:string);
procedure CCmp(MSize : SizeType;Source,Dest:string);
procedure CClr(MSize : SizeType;Dest:string);
procedure DoBranch(Cond,Labl:longint);
procedure PlaceLabelAct(Labl:longint);
procedure PlaceForgetLabelAct(Labl:longint);
function GBTrunc(D:double):longint;
function RealToLongintfR: longint;

implementation

type
ExtDescType = record
ExtCnt: word;
Mode,Reg : byte;
ExtWords : array[1..6] of word;
ImmVal : longint;
end;

var
DoSubtract: boolean;

function SPV: integer;
var t,i : integer;
begin
  t := SP;
  for i := SP downto 0 do
    if CStack[i].SType = OldDead then dec(t);
  SPV := t;
end;

function LoWord(l:longint):word;
begin
  LoWord := 1 and $ffff;
end;
function HiWord(l:longint):word;
begin
  HiWord := ((l and $ffff0000) div $10000) and $ffff;
end;

procedure GencodeInit;
var i:integer;
begin
  DoSubtract := false;
  SP := -1; SPm1 := -2; Spm2 := -3;
  for i := 0 to MaxSp do
    with CStack[i] do
      begin
        SType := Empty; {set up a default stack entry}
        IsReal := false;
        BaseAddr := NoBase;
        BLevel := 0;
      end;
end;

procedure CompressStack;
var i,j : integer;
  Strm1,Strp1 : string[35];
begin
  i := SP-1;
  while (i >= 0) and (CStack[i].SType <> OldDead) do dec(i);
  if i >= 0 then begin
    for j := i to SP-1 do begin
      CStack[i] := CStack[i+1];
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str(i,Strm1);
        str(i+1,Strp1);
        if CStack[i].IsReal then
          CListOutCftnove.x fp'+Strp1+','fp'+Strm1,2,
          $f200,$0000 + $400*(i+1) + $80*i,0,0)
        else
          CMove(L,'d'+ Strp1,'d'+ Strm1);
      end;
    end;
    dec(SP);
    nTop := SP;
    nNext := SP-1;
    while (nNext > -1) and (CStack[nNext].SType = OldDead) do dec(nNext);
    Spm1 := nNext;
    Spm2 := Spm1 - 1;
    while (Spm2 > -1) and (CStack[Spm2].SType = OldDead) do dec(Spm2);
    str(nTop,TopStr);
    str(nNext,NextStr);
  end;
end;
procedure incSP;
begin
{check for a condition code on top of the stack first}
if SP >= 0 then
  if CStack[SP].SType=CondC then begin
    case CStack[SP].CondField of
      CondCC : CListOut('scb d'+TopStr,1,$54c0+nTop,0,0,0);
      CondCS : CListOut('scb d'+TopStr,1,$55c0+nTop,0,0,0);
      CondEQ : CListOut('seq.b d'+TopStr,1,$57c0+nTop,0,0,0);
      CondGE : CListOut('sge.b d'+TopStr,1,$5cc0+nTop,0,0,0);
      CondGT : CListOut('sgt.b d'+TopStr,1,$5ec0+nTop,0,0,0);
      CondHI : CListOut('shi.b d'+TopStr,1,$52c0+nTop,0,0,0);
      CondLE : CListOut('sle.b d'+TopStr,1,$5fc0+nTop,0,0,0);
      CondLS : CListOut('sls.b d'+TopStr,1,$53c0+nTop,0,0,0);
      CondLT : CListOut('slt.b d'+TopStr,1,$5dc0+nTop,0,0,0);
      CondMI : CListOut('smi.b d'+TopStr,1,$5bc0+nTop,0,0,0);
      CondNE : CListOut('sne.b d'+TopStr,1,$56c0+nTop,0,0,0);
      CondPL : CListOut('spl.b d'+TopStr,1,$5ac0+nTop,0,0,0);
      CondVC : CListOut('svc.b d'+TopStr,1,$58c0+nTop,0,0,0);
      CondVS : CListOut('svs.b d'+TopStr,1,$59c0+nTop,0,0,0);
      CondT,
      CondF : begin
        CStack[SP].SType := ConstOnly;
        CStack[SP].ConstVal := 1 - CStack[SP].CondField;
        end;
      end;
    else FlagError(-1000);
  end;
end;
if CStack[SP].SType=CondC then begin
  CListOut('neg.b d'+TopStr,1,$4400+nTop,0,0,0);
  CListOut('extb.l d'+TopStr,1,$49c0+nTop,0,0,0);
  CStack[SP].SType := RegOnly;
end;
end;
if SP = MaxSp then CompressStack;
if SP = MaxSp then FlagError(259);
inc(Sp);
with CStack[SP] do
begin
  SType := Empty; {set up a default stack entry}
  IsReal := false;
  BaseAddr := NoBase;
  CondField := $ff;
  ConstVal := 0;
end;
nTop := SP;
nNext := SP-1;
while (nNext > -1) and (CStack[nNext].SType = OldDead) do dec(nNext);
SPm1 := nNext;
SPm2 := SPm1 - 1;
while (SPm2 > -1) and (CStack[SPm2].SType = OldDead) do dec(SPm2);
str(nTop,TopStr);
procedure decSP;
var i:integer; NewDeadExists : boolean;
begin
  if CStack[SP].SType = CondC then FlagError(-1002);
  NewDeadExists := false;
  for i := SP downto 0 do if CStack[i].SType=NewDead then
    NewDeadExists := true;
  if not NewDeadExists then dec(SP);
  while (SP > -1) and (CStack[SP].SType = OldDead) do dec(SP);
  for i := SP downto 0 do
    if CStack[i].SType = NewDead then CStack[i].SType := OldDead;
  nTop := SP;
  nNext := SP-1;
  while (nNext > -1) and (CStack[nNext].SType = OldDead) do dec(nNext);
  SPm1 := nNext;
  SPm2 := SPm1 - 1;
  while (SPm2 > -1) and (CStack[SPm2].SType = OldDead) do dec(SPm2);
  str(nTop,TopStr);
  str(nNext,NextStr);
end;

procedure Simplify;
var i,j1,j2:integer;
begin
  {$R-}
  if (SP > = 0 ) and (SP <= MaxSp) then
  {$R+}
  begin
    j1 := SP - 1;
    if j1 < 0 then j1 := 0;
    j2 := SP;
    for i := j1 to j2 do
      with CStack[i] do
        case SType of
          ConstPlusReg : if ConstVal=0.0 then SType := RegOnly;
          ConstTimesReg : if ConstVal=0.0 then SType := ConstOnly
                          else if ConstVal = 1.0 then SType := RegOnly;
        end;
  end;
end;

procedure ExpandCC(entry:byte);
var EStr: string[5];
begin
  if CStack[entry].SType=CondC then begin
    str(entry,EStr);
    case CStack[entry].CondField of
      CondCC : CListOut('scc.b d'+EStr,l,$54c0+entry,0,0,0);
end;
CondCS : CListOut('scs.b d' + EStr, 1, $55c0 + entry, 0, 0, 0);
CondEQ : CListOut('seq.b d' + EStr, 1, $57c0 + entry, 0, 0, 0);
CondGE : CListOut('sge.b d' + EStr, 1, $5cc0 + entry, 0, 0, 0);
CondGT : CListOut('sgt.b d' + EStr, 1, $5ec0 + entry, 0, 0, 0);
CondHI : CListOut('shi.b d' + EStr, 1, $52c0 + entry, 0, 0, 0);
CondLE : CListOut('sle.b d' + EStr, 1, $5fc0 + entry, 0, 0, 0);
CondLS : CListOut('sls.b d' + EStr, 1, $53c0 + entry, 0, 0, 0);
CondLT : CListOut('slt.b d' + EStr, 1, $5dc0 + entry, 0, 0, 0);
CondMI : CListOut('smi.b d' + EStr, 1, $5bc0 + entry, 0, 0, 0);
CondNE : CListOut('sne.b d' + EStr, 1, $56c0 + entry, 0, 0, 0);
CondPL : CListOut('spl.b d' + EStr, 1, $5ac0 + entry, 0, 0, 0);
CondVC : CListOut('svc.b d' + EStr, 1, $58c0 + entry, 0, 0, 0);
CondVS : CListOut('svs.b d' + EStr, 1, $59c0 + entry, 0, 0, 0);
CondT,

CondF : begin
    CStack[SP].SType := ConstOnly;
    CStack[SP].ConstVal := 1 - CStack[SP].CondField;
end;
else FlagError(-1001);
end;

if CStack[SP].SType = CondC then begin
    CListOut('neg.b d' + EStr, 1, $4400 + entry, 0, 0, 0);
    CListOut('extb.l d' + EStr, 1, $49c0 + entry, 0, 0, 0);
    CStack[entry].SType := RegOnly;
    end;
end;

procedure negate;
var
    L : longint;
begin
    Simplify;
    if CStack[SP].BaseAddr <> NoBase then FlagError(-20); {fix the compiler}
    with CStack[SP] do
        case SType of
            Empty:  FlagError(-21);
            ConstOnly : ConstVal := -ConstVal;
            RegOnly : if IsReal then
                CListOut('fneg.x fp' + TopStr, 2, $f200, $001a + $400*nTop + $80*nTop, 0, 0)
                else
                CListOut('neg.l d' + TopStr, 1, $4480 + nTop, 0, 0, 0);
            ConstPlusReg : begin
                if IsReal then
                    CListOut('fneg.x fp' + TopStr, 2, $f200, $001a + $400*nTop + $80*nTop, 0, 0)
                else
                    CListOut('neg.l d' + TopStr, 1, $4480 + nTop, 0, 0, 0);
                ConstVal := -ConstVal;
            end;
ConstTimesReg : ConstVal := -ConstVal;
end;
end;

procedure i2fpTop;
var
  Num : longint;
  NumStr : string[12];
begin
  if CStack[SP].BaseAddr <> NoBase then FlagError(-22); {fix the compiler}
  with CStack[SP] do begin
    if SType in [RegOnly,ConstPlusReg,ConstTimesReg] then
      CListOut(c'move.l d'+TopStr+'+,fp'+TopStr+',2,$f200 + nTop,
               $4000 + $80*nTop,0,0);
    IsReal := true;
  end;
end;

procedure i2fpNext;
var
  Num : longint;
  NumStr : string[12];
begin
  if CStack[SPm1].BaseAddr <> NoBase then FlagError(-23); {fix the compiler}
  with CStack[SPm1] do begin
    if SType in [RegOnly,ConstPlusReg,ConstTimesReg] then
      CListOut(c'move.l d'+NextStr+'+,fp'+NextStr+',2,$f200 + nNext,
               $4000 + $80*nNext,0,0);
    IsReal := true;
  end;
end;

procedure SetTypes;
begin
  if CStack[SP].IsReal or CStack[SPm1].IsReal then
    begin
      if not CStack[SP].IsReal then i2fpTop;
      if not CStack[SPm1].IsReal then i2fpNext;
    end;
end;

procedure SaveStk;
var i : integer; Num : string[4];
begin
  if SP >= 7 then begin
    for i := SP downto SP-7 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then
        begin
          str((i mod 8),Num);
          if CStack[i].IsReal then
            
  
```
begin
  if SP >= 7 then begin
    for i := SP-7 to SP do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := 0 to SP do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := SP downto 0 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := SP downto 0 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end;

procedure RestoreStk;
var i : integer; Num : string[4];
begin
  if SP >= 7 then begin
    for i := SP-7 to SP do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := 0 to SP do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := SP downto 0 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := SP downto 0 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end else begin
    for i := SP downto 0 do
      if CStack[i].SType in [RegOnly,ConstPlusReg,ConstTimesReg] then begin
        str((i mod 8),Num);
        if CStack[i].IsReal then
          CLListOutCfinove.d (a7)'+,fp'+Num+',2,$f227,$7400 + $80*(i mod 8),0,0)
        else
          CMove(L,'d'+Num,'- (a7)');
      end;
  end;

procedure GetFullAddr(entry:byte);
var i : integer;
  EntryStr : string[4];
  Offs : longint;
  ARReg,DReg : integer;
begin
  Simplify;

Dreg := entry mod 8; \{assume data register will be used\}
Areg := -1; \{assume address register will not be used\}
with CStack[entry] do begin
  if not(SType in [ConstOnly, RegOnly, ConstPlusReg, Empty]) then
    FlagError(-24);
  if IsReal then FlagError(-25);
  if (BaseAddr = VarAddr) then begin
    if (BLevel = 0) or ((ProcessLevelPtr <> 0)
      AND (BLevel = ProcessLevel[ProcessLevelPtr])) then
      AReg := 6
    else begin
      if not (SType in [ConstOnly, ConstPlusReg]) then begin
        \{prepare to have integer constant added on\}
        ConstVal := 0.0;
        case SType of
          RegOnly : SType := ConstPlusReg;
          Empty : SType := ConstOnly;
        end;
      end;
      ConstVal := ConstVal + 4.0;
      if BLevel = BlockLevel then AReg := 5
      else begin
        AReg := 3;
        if not A3Valid OR (A3AddrType <> VarAddr) OR (A3BlockLevel <> BLevel)
          then begin
            if A3Valid AND (A3AddrType = VarAddr) AND (A3BlockLevel > BLevel)
              then for i := BLevel to A3BlockLevel do CMove(L,'(a3)Va3')
                else begin
                  CMove(L,'(a5)Va3');
                  for i := BLevel to BlockLevel-2 do CMove(L,'(a3)Va3');
                end;
            A3Valid := true;
            A3AddrType := VarAddr;
            A3BlockLevel := BLevel;
          end;
      end;
  end;
end; \{if (BaseAddr = VarAddr)\}
if (BaseAddr = ParamAddr) then begin
  if BLevel = BlockLevel then AReg := 4
  else begin
    AReg := 3;
    if not A3Valid OR (A3AddrType <> ParamAddr) OR (A3BlockLevel <> BLevel)
      then begin
        if A3Valid AND (A3AddrType = ParamAddr) AND (A3BlockLevel > BLevel)
          then for i := BLevel to A3BlockLevel do CMove(L,'-4(a3)Va3')
            else begin
              CMove(L,'-4(a4)Va3');
              for i := BLevel to BlockLevel-2 do CMove(L,'-4(a3)Va3');
            end;
        end;
  end;
A3Valid := true;
A3AddrType := ParamAddr;
A3BlockLevel := BLevel;
end;
end; {if (BaseAddr = ParamAddr)}
{at this point, AReg is calculated}
{Now, determine appropriate values for Offs and DReg}
case SType of
  ConstOnly: begin
    Offs := GBTrunc(ConstVal);
    DReg := -1;
    end;
  RegOnly: begin
    Offs := 0;
    if AReg=-1 then begin
      AReg := 3;
      A3Valid := false;
      str(entry mod 8,EntryStr);
      CMove(L,'d'+ EntryStr,'a3');
      DReg := -1;
    end;
    end;
  ConstPlusReg: begin
    Offs := GBTrunc(ConstVal);
    if AReg=-1 then begin
      AReg := 3;
      A3Valid := false;
      str(entry mod 8,EntryStr);
      CMove(L,'d'+ EntryStr,'a3');
      DReg := -1;
    end;
    end;
  ConstTimesReg: FlagError(-26);
  Empty: begin
    if AReg=-1 then FlagError(-27); {no address}
    Offs := 0; {no offset used}
    DReg := -1; {no data register used}
    end;
end; {with CStack[entry] do}
{now, build up AddrStr}
AddrStr := "; {start with null string}
if (AReg=-1) and (DReg=-1) then begin
  EAMode := 7;
  if abs(Offs) < $7fff then begin
    EAReg := 0;
    NumExtWords := 1;
    Ext1 := LoWord(Offs);
  end else begin
    EAReg := 1;
    NumExtWords := 2;
  end;
end;
Ext1 := HiWord(Offs);
Ext2 := LoWord(Offs);
end;
end;
if (AReg <> -1) and (DReg <> -1) then begin
if Offs=0 then begin
EAreg := AReg;
EAMode := 6;
NumExtWords := 1;
Ext1 := $0800 + DReg*$1000;
end else begin
if abs(Offs) < 127 then begin
EAreg := AReg;
EAMode := 6;
NumExtWords := 1;
Ext1 := $0800 + DReg*$1000 + (Offs and $ff);
end else begin
if abs(Offs) < $7fff then begin
EAreg := AReg;
EAMode := 5;
NumExtWords := 1;
Ext1 := LoWord(Offs);
end else begin
EAreg := AReg;
EAMode := 6;
NumExtWords := 3;
Ext1 := $0930 + DReg*$1000;
Ext2 := HiWord(Offs);
Ext3 := LoWord(Offs);
end;
end;
end;
end;
if (AReg <> -1) and (DReg = -1) then begin
if Offs=0 then begin
EAReg := AReg;
NumExtWords := 0;
EAMode := 2;
end else begin
if abs(Offs) < $7fff then begin
EAReg := AReg;
EAMode := 5;
NumExtWords := 1;
Ext1 := LoWord(Offs);
end else begin
EAReg := AReg;
EAMode := 6;
NumExtWords := 3;
Ext1 := $0170;
Ext2 := HiWord(Offs);
Ext3 := LoWord(Offs);
end;
end;
end;
if Offs <> 0 then str(Offs,AddrStr);
if (AReg <> -1) then begin
if (Offs< -32768) or (Offs> 32767) then AddrStr := '(' + AddrStr + ',a'
else AddrStr := AddrStr + '(a';
str(Areg,EntryStr);
AddrStr := AddrStr + EntryStr;
end;
if (DReg <> -1) then begin
AddrStr := AddrStr + ',d';
str(DReg,EntryStr);
AddrStr := AddrStr + EntryStr + ')';
end else begin
if pos(',',AddrStr) <> 0 then
  AddrStr := AddrStr + ');
end;
end;

procedure AddBaseAddr(entry:byte);
var RegNum : byte;
  RegStr,DRegStr : string[4];
  i : integer;
begin
  Simplify;
  str(entry mod 8,DRegStr);
  with CStack[entry] do begin
    if not(SType in [ConstOnly,RegOnly,ConstPlusReg,Empty]) then
      FlagError(-28);
    if (BaseAddr = VarAddr) then begin
      if (BLevel = 0) OR ((ProcessLevelPtr <> 0)
      AND (BLevel = ProcessLevel[ProcessLevelPtr]))
        then RegNum := 6
    else begin
      ConstVal := 0.0;
      case SType of
        RegOnly : SType := ConstPlusReg;
        Empty : SType := ConstOnly;
      end;
    end;
    ConstVal := ConstVal + 4.0;
    if BLevel=BlockLevel then RegNum := 5
  else begin
    RegNum := 3;
    if not A3Valid OR (A3AddrType<>VarAddr) OR (A3BlockLevel<>BLevel)
      then begin
        if A3Valid AND (A3AddrType=VarAddr) AND (A3BlockLevel>BLevel)
          then for i := BLevel to A3BlockLevel do CMove(L,(\'a3\'),\'a3\')
else begin
  CMove(L,'(a5)','a3');
  for i := BLevel to BlockLevel-2 do CMove(L,'(a3)','a3');
end;
end;
A3Valid := true;
A3AddrType := VarAddr;
A3BlockLevel := BLevel;
end;
end; {if (BaseAddr = VarAddr)}
if (BaseAddr=ParamAddr) then begin
  if BLevel=BlockLevel then RegNum := 4
  else begin
    RegNum := 3;
    if not A3Valid OR (A3AddrType<>ParamAddr) OR (A3BlockLevel<>BLevel)
      then begin
        if A3Valid AND (A3AddrType=ParamAddr) AND (A3BlockLevel>BLevel)
          then for i := BLevel to A3BlockLevel do CMove(L,-4(a3),'a3')
        else begin
          CMove(L,-4(a4),'a3');
          for i := BLevel to BlockLevel-2 do CMove(L,-4(a3),'a3');
        end;
      end;
    A3Valid := true;
    A3AddrType := ParamAddr;
    A3BlockLevel := BLevel;
  end;
end; {if (BaseAddr = ParamAddr)}
if (BaseAddr=VarAddr) or (BaseAddr=ParamAddr) then begin
  str(RegNum,RegStr);
  if IsReal then FlagError(-29) {fix this}
  else begin
    if SType in [RegOnly,ConstPlusReg] then begin
      CListOut('add1 '+'a'+RegStr+',d'+DRegStr,l,
               $d088 + RegNum + $200*(entry mod 8),0,0,0);
    end else begin
      CMove(L,'a'+RegStr,'d'+DRegStr);
      case SType of
        Empty : SType := RegOnly;
        ConstOnly : SType := ConstPlusReg;
        ConstTimesReg : FlagError(-30);
      end;
      end;
    end;
  end;
  BaseAddr := NoBase;
end; {with CSta...}
end;

procedure AddlnConstant(entry:byte);
begin
Simplify;
str(entry mod 8,RegStr);
with CStack[entry] do
  case SType of
    ConstOnly : if IsReal then begin
      str(ConstVal,NumStr);
      alias1 := ConstVal;
      CListOut('fmove.d #'+NumStr+',fp'+RegStr,4,
        $f23c,$5400 + $80*(entry mod 8),
        alias2[3],alias2[2]);
      CListOut('a',2,alias2[1],alias2[0],0,0);
      SType := RegOnly;
    end else begin
      IntTmp := GBTrunc(ConstVal);
      str(IntTmp,NumStr);
      CMove(L,'#'+NumStr,'d'+RegStr);
      SType := RegOnly;
    end;
    ConstPlusReg : if IsReal then begin
      str(ConstVal,NumStr);
      alias1 := ConstVal;
      CListOut('fadd.d #'+NumStr+',fp'+RegStr,4,
        $f23c,$5422 + $80*(entry mod 8),
        alias2[3],alias2[2]);
      CListOut('a',2,alias2[1],alias2[0],0,0);
      SType := RegOnly;
    end else begin
      IntTmp := GBTrunc(ConstVal);
      str(IntTmp,NumStr);
      CMove(L,'#'+NumStr,'d'+RegStr);
      SType := RegOnly;
    end;
    ConstTimesReg : {do nothing} ;
  end;
end;

procedure ClearAddIns(entry:byte);
var LStr,NumStr,RegStr : string[15];
  IntTmp : longint;
  alias1 : double;
  alias2 : array[0..3] of word absolute alias1;
begin
  {do nothing}
end;
begin
Simplify;
str(entry mod 8,RegStr);
with CStack[entry] do case SType of
ConstOnly,
RegOnly : {do nothing};
ConstPlusReg : if IsReal then begin
str(ConstVal,NumStr);
alias1 := ConstVal;
CListOut('fadd.d #'+NumStr+',fp'+RegStr,4,
$s23c,$5422 + $80*(entry mod 8),
alias2[3],alias2[2]);
CListOut('"2,alias2[1],alias2[0],0,0);
SType := RegOnly;
end else begin
IntTmp := GBTrunc(ConstVal);
str(IntTmp,NumStr);
CListOut('addi.l #'+NumStr+',d'+RegStr,3,
$0680 + (entry mod 8),HiWord(IntTmp),
LoWord(IntTmp),0);
SType := RegOnly;
end;
ConstTimesReg : begin {do nothing} end;
end;
end;

procedure MultInConstant(entry:byte);
var LStr, NumStr, RegStr : string[35];
IntTmp : longint;
MulVal,
ShiftCount : longint;
ShiftStr : string[5];
alias1 : double;
alias2 : array[0..3] of word absolute alias1;
begi
Simplify;
str(entry mod 8,RegStr);
with CStack[entry] do case SType of
Empty,
ConstOnly,
RegOnly,
ConstPlusReg : {do nothing} ;
ConstTimesReg :
if IsReal then begin
str(ConstVal,NumStr);
alias1 := ConstVal;
CListOut('fmul.d #'+NumStr+',fp'+RegStr,4,
$s23c,$5423 + $80*(entry mod 8),
alias2[3],alias2[2]);
CListOut("2,alias2[1],alias2[0],0,0);
SType := RegOnly;
end else begin
IntTmp := GBTrunc(ConstVal);
ShiftCount := 0;
while (IntTmp and 1) = 0 do begin
IntTmp := IntTmp shr 1;
inc(ShiftCount);
end;
if IntTmp=1 then begin
if ShiftCount <> 0 then begin
while ShiftCount > 8 do begin
CListOut(asl.1 #8,d'+RegStr,1,$e180+
(entry mod 8),0,0,0);
dec(ShiftCount,8);
end;
str(ShiftCount,ShiftStr);
CListOut(asl.1 #'+ShiftStr+',d'+RegStr,1,
$e180+$200*(ShiftCount mod 8) + (entry mod 8),
0,0,0);
end;
end else begin
IntTmp := GBTrunc(ConstVal);
str(IntTmp,NumStr);
CListOut(' muls.l #'+NumStr+',d'+RegStr,4,
$4c3c,$0800 + $1000*(entry mod 8),
HiWord(IntTmp),LoWord(IntTmp));
end;
SType := RegOnly;
end;
end;
end;

procedure DoMoveQ(reg:byte;val:byte);
var RegStr,ValStr : string[4];
begin
str(reg,RegStr);
RegStr := 'd' + RegStr;
str(val,ValStr);
CListOut('moveq.l #'+ValStr+','+RegStr,1,$7000 + $200*reg + val,0,0,0);
end;

procedure Trap2;
const
Trap2Code = $4e42;
begin
CListOut('trap #2,1,Trap2Code,0,0,0);
end;

procedure OSFunc(N:byte);
begin
  DoMoveQ(0,N);
  Trap2;
end;

procedure PushEA;
begin
  CListOut('pea.1 ' + AddrStr,1+NumExtWords,
            $4840 + $8+EAMode + EReg,Ext1,Ext2,Ext3);
end;

procedure ScanOperand(s:string;var Ext:ExtDescType;MSize:SizeType);

procedure Flip;
begin
  writeln;
  writeln('Error in Operand : ',s);
  halt(500);
end;

var
  Value,P,Base,DigValue,i : longint;
  AReg,DReg : integer;
  Negate : boolean;

procedure GetValue;
begin
  Value := 0;
  if not (s[P] in ['0'..'9','$','']) then Exit;
  Negate := s[P] = '$';
  if Negate then inc(P);
  if s[P]='$' then begin
    Base := 16;
    inc(P);
  end else Base := 10;
  Value := 0;
  while (P <= length(s)) and (s[P] in ['0'..'9','A'..'F']) do begin
    case UpCase(s[P]) of
      '0'..'9' : DigValue := ord(s[P]) - ord('0');
      'A'..'F' :
        begin
          if Base <> 16 then Flip;
          DigValue := ord(upcase(s[P])) - ord('A') + 10;
        end;
    end;
    Value := Base*Value + DigValue;
    inc(P);
  end;
  if Negate then Value := -Value;
end;
begin
while pos(' ',s) <> 0 do delete(s,pos(' ',s),1);
for i := 1 to length(s) do s[i] := UpCase(s[i]);
if s='" then Flip;
if s[1] in ['A','D'] then begin {modes 0&1}
  if length(s) <> 2 then Flip;
  if (s[1]= 'A') and (MSize=B) then Flip;
  if not (s[2] in ['0','7']) then Flip;
  with Ext do begin
    if s[1]= 'A' then mode := 1 else mode := 0;
    Reg := ord(s[2]) - ord('0');
    ExtCnt := 0;
  end;
  Exit;
end;
end;

if (copy(s,1,2) = '(A' and (copy(s,4,5)=')') then begin {mode 2}
  if not (s[3] in ['0','7']) then Flip;
  if length(s) <> 4 then Flip;
  with Ext do begin
    mode := 2;
    Reg := ord(s[3]) - ord('0');
    ExtCnt := 0;
  end;
  Exit;
end;
end;

if (copy(s,1,2) = '(A' and (copy(s,4,5)=')+) then begin {mode 3}
  if not (s[3] in ['0','7']) then Flip;
  with Ext do begin
    mode := 3;
    Reg := ord(s[3]) - ord('0');
    ExtCnt := 0;
  end;
  Exit;
end;
end;

if (copy(s,1,3) = '-(A' and (copy(s,5,5)=')') then begin {mode 4}
  if not (s[4] in ['0','7']) then Flip;
  with Ext do begin
    mode := 4;
    Reg := ord(s[4]) - ord('0');
    ExtCnt := 0;
  end;
  Exit;
end;
end;

if s[1]= '"' then begin {mode 7.4}
P := 2;
GetValue;
if \( P \leq \text{length}(s) \) then Flip;

with Ext do begin
  ImmVal := Value;
  Mode := 7;
  Reg := 4;
  case MSize of
    L : begin
      ExtCnt := 2;
      ExtWords[1] := HiWord(Value);
      ExtWords[2] := LoWord(Value);
    end;
    W : begin
      ExtCnt := 1;
      ExtWords[1] := LoWord(Value);
    end;
    B : begin
      ExtCnt := 1;
      ExtWords[1] := $ff \text{ and } LoWord(Value);
    end;
  end;
end;
Exit;
end;

{if we reach this point, its one of the "other" modes - some combination
of base address, address register, and data register}

\( P := 1; \)
GetValue;

if \( P > \text{length}(S) \) then begin {modes 7.0 & 7.1} with Ext do begin
  Mode := 7;
  if (Value \geq -32768) and (Value \leq 32767) then begin
    Reg := 0;
    ExtCnt := 1;
    ExtWords[1] := LoWord(Value);
  end else begin
    Reg := 1;
    ExtCnt := 2;
    ExtWords[1] := HiWord(Value);
    ExtWords[2] := LoWord(Value);
  end;
end;
Exit;
end;

if (P=1) and (s[1]='\(') then begin
  P := 2;
  GetValue;
  if (P<>2) and (s[P] <> ',') then Flip;

if P<>2 then inc(P);
end else begin
  if s[P] <> 'C' then Flip;
  inc(P);
end;

if not (s[P] in ['A','D']) then Flip;

{we now have Value}

AReg := -1; DReg := -1;
while (s[P] <> ')') and (P <= length(s)) do begin
case UpCase(s[P]) of
  'A' : begin
    if AReg <> -1 then Flip;
    if not (s[P+1] in ['0'..'7']) then Flip;
    AReg := ord(s[P+1]) - ord('0');
    inc(P,2);
  end;
  'D' : begin
    if DReg <> -1 then Flip;
    if not (s[P+1] in ['0'..'7']) then Flip;
    if s[P+2] <> '.' then Flip;
    if UpCase(s[P+3]) <> 'L' then Flip;
    DReg := ord(s[P+1]) - ord('0');
    inc(P,4);
  end;
else Flip;
end;
if s[P] =',' then begin
  inc(P);
  if s[P] = ')' then Flip;
end else if s[P] <> ')' then Flip;
end;
if P > length(S) then Flip;

{we have now determined Value, DReg, and AReg}

if (AReg=-1) and (DReg=-1) then Flip;
if (AReg=-1) and (DReg<>-1) then Flip;
if (AReg <> -1) and (DReg=-1) then begin
  if Value=0 then begin {offset=0 means use mode 2}
    with Ext do begin
      Mode := 2;
      Reg := AReg;
      ExtCnt := 0;
      end;
Exit;
end;
end;
if (Value >= -32768) and (Value <= 32767) then begin (mode 5)
    with Ext do begin
        Mode := 5;
        Reg := AReg;
        ExtCnt := 1;
        ExtWords[1] := LoWord(Value);
    end;
    Exit;
end;
end;

{if we get here, we have a long word offset combined with AReg}
with Ext do begin
    Reg := AReg;
    Mode := 6;
    ExtCnt := 3;
    ExtWords[1] := $0170;
    ExtWords[2] := HiWord(Value);
    ExtWords[3] := LoWord(Value);
end;
Exit;
end;
end;

{if we get here, AReg <> -1 and DReg <> -1}
{Value can be anything, including zero}

if (Value >= -128) and (Value <= 127) then begin (brief format extension)
    with Ext do begin
        Reg := AReg;
        Mode := 6;
        ExtCnt := 1;
        ExtWords[1] := $0800 + DReg*$1000 + ($ff and Value);
    end;
    Exit;
end;
end;

{if we get here, AReg <> -1 and DReg <> -1}
{Value can be either word or long}

if (Value >= -32768) and (Value <= 32767) then begin
    with Ext do begin
        Reg := AReg;
        Mode := 6;
        ExtCnt := 2;
        ExtWords[1] := $0920 + DReg*$1000;
        ExtWords[2] := LoWord(Value);
    end;
    Exit;
end;
end;

{if we get here, AReg <> -1 and DReg <> -1}
{Value must be long}
with Ext do begin
  Reg := AReg;
  Mode := 6;
  ExtCnt := 3;
  ExtWords[1] := $0930 + DReg*$1000;
  ExtWords[2] := HiWord(Value);
  ExtWords[3] := LoWord(Value);
end;
end;

procedure CMove(MSize : SizeType;Source,Dest:string);
var SExt,DExt : ExtDescType;
instr : string;
InstrWord : word;
CExtWords : array[1..10] of word;
CExtCnt,i : word;
begin
  ScanOperand(Source,SExt,MSize);
  ScanOperand(Dest,DExt,MSize);
  if (DExt.Mode=0) and (SExt.Mode=7) and (SExt.Reg=4) and
    (SExt.ImmVal >= -128) and (SExt.ImmVal <= 127) then begin
    instr := 'moveq.';
    InstrWord := $7000 + $200*DExt.Reg + ($ff and SExt.ImmVal);
    SExt.ExtCnt := 0;
  end else begin
    if DExt.Mode=1 then begin
      instr := 'movea.';
      if not(MSize in [L,W]) then begin
        writeln; writeln(' Error in Move to Addr Reg : ',Source); halt(500);
      end;
    end else instr := 'move.';
    case MSize of
      L : InstrWord := $2000;
      W : InstrWord := $3000;
      B : InstrWord := $1000;
      end;
    with SExt do
      InstrWord := InstrWord + Reg + $8*Mode;
    with DExt do
      InstrWord := InstrWord + $200*Reg + $40*Mode;
  end;
  case MSize of
    L : instr := instr + 'l';
    W : instr := instr + 'w';
    B : instr := instr + 'b';
  end;
  while length(instr) < 8 do instr := instr + ' ';
  instr := instr + Source + ',' + Dest;
CExtCnt := 0;
for i := 1 to SExt.ExtCnt do begin
  inc(CExtCnt);
  CExtWords[CExtCnt] := SExt.ExtWords[i];
end;
for i := 1 to DExt.ExtCnt do begin
  inc(CExtCnt);
  CExtWords[CExtCnt] := DExt.ExtWords[i];
end;
if 1+CExtCnt <= 4 then
  CListOut(instr,1+CExtCnt,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3])
else begin
  CListOut(instr,4,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3]);
  CListOut(',',1+CExtCnt-4,CExtWords[4],CExtWords[5],CExtWords[6],0);
end;
end;

procedure CAdd(MSize : SizeType;Source,Dest:string);
var SExt,DExt : ExtDescType;
instr : string;
InstrWord : word;
CExtWords : array[1..10] of word;
CExtCnt,i : word;
begin
  ScanOperand(Source,SExt,MSize);
  ScanOperand(Dest,DExt, MSize);
  if (MSize=B) and ((DExt.Mode=1) or (SExt.Mode=1)) then FlagError(804);
  if (DExt.Mode=7) and (DExt.Reg=4) then FlagError(803);
  if (SExt.Mode=7) and (SExt.Reg=4) and (DExt.Mode <> 1) then begin
    if (SExt.ImmVal >= 1) and (SExt.ImmVal <= 8) then begin
      instr := 'addq.';
      InstrWord := $5000;
      case MSize of
        L : InstrWord := InstrWord + $80;
        W : InstrWord := InstrWord + $40;
      end;
      InstrWord := InstrWord + $200 * (SExt.ImmVal and $07);
      SExt.ExtCnt := 0;
    end else begin
      instr := 'addi.';
      InstrWord := $0600;
      case MSize of
        L : InstrWord := InstrWord + $80;
        W : InstrWord := InstrWord + $40;
      end;
    end;
  end else begin
    instr := 'addl.';
    InstrWord := $0600;
    case MSize of
      L : InstrWord := InstrWord + $80;
      W : InstrWord := InstrWord + $40;
    end;
  end;
end;
if DExt.Mode=1 then begin
  if (SExt.Mode=7) and (SExt.Reg=4) and (SExt.ImmVal >= 1) and
  (SExt.ImmVal <= 8) then begin
    instr := 'addq.';
    InstrWord := $5000;
    case MSize of
      L : InstrWord := InstrWord + $80;
      W : InstrWord := InstrWord + $40;
    end;
    InstrWord := InstrWord + $200 * (SExt.ImmVal and $07);
    SExt.ExtCnt := 0;
  end else begin
    instr := 'adda.';
    InstrWord := $d000;
    case MSize of
      L : InstrWord := InstrWord + $1c0;
      W : InstrWord := InstrWord + $0c0;
    end;
    InstrWord := InstrWord + $200*DExt.Reg;
  end;
end;
end;

if (DExt.Mode <> 1) and not((SExt.Mode=7) and (SExt.Reg=4)) then begin
  instr := 'add.';
  if DExt.Mode = 0 then begin
    InstrWord := $d000;
    InstrWord := InstrWord + $200 * DExt.Reg;
  end else begin
    if SExt.Mode <> 0 then FlagError(804);
    InstrWord := $d100;
    InstrWord := InstrWord + $200 * SExt.Reg;
  end;
  case MSize of
    L : InstrWord := InstrWord + $80;
    W : InstrWord := InstrWord + $40;
  end;
end;
end;

case MSize of
  L : instr := instr + 'l';
  W : instr := instr + 'w';
  B : instr := instr + 'b';
end;
while length(instr) < 8 do instr := instr + ' ';
instr := instr + Source + ',' + Dest;
CExtCnt := 0;
for i := 1 to SExt.ExtCnt do begin
  inc(CExtCnt);
  CExtWords[CExtCnt] := SExt.ExtWords[i];
end;
for i := 1 to DExt.ExtCnt do begin
  inc(CExtCnt);
  CExtWords[CExtCnt] := DExt.ExtWords[i];
end;
if DoSubtract then begin
  instr := 'sub' + copy(instr,4,255);
  if copy(instr,1,4) = 'subq' then InstrWord := InstrWord + $100
  else begin
    if copy(instr,1,4) = 'subi' then InstrWord := InstrWord - $200
    else InstrWord := InstrWord - $4000;
  end;
end;
if 1+CExtCnt <= 4 then
  CListOut(instr,1+CExtCnt,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3])
else begin
  CListOut(instr,4,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3]);
  CListOut("","1+CExtCnt-4,CExtWords[4],CExtWords[5],CExtWords[6],0);
end;
end;

procedure CSub(MSize : SizeType; Source,Dest:string);
begin
  DoSubtract := true;
  CAdd(MSize,Source,Dest);
  DoSubtract := false;
end;

procedure CLea(Source,Dest:string);
var SExt,DExt : ExtDescType;
  InstrWord : word;
begin
  ScanOperand(Source,SExt,L);
  ScanOperand(Dest,DExt,L);
  if DExt.Mode <> 1 then FlagError(800);
  if (SExt.Mode < 5) and (SExt.Mode <> 2) then FlagError(801);
  if (SExt.Mode=7) and (SExt.Reg=4) then FlagError(802);
  InstrWord := $41c0;
  InstrWord := InstrWord + $200*DExt.Reg;
  with SExt do begin
    CListOut('lea.l','+Source+',',+Dest,1+ExtCnt,InstrWord,ExtWords[1],
      ExtWords[2],ExtWords[3]);
    if ExtCnt > 3 then
      CListOut("","ExtCnt-3,InstrWord,ExtWords[4],
        ExtWords[5],ExtWords[6]);
procedure CClr(MSize:SizeType;Dest:string);
var DExt : ExtDescType;
   instr : string;
   InstrWord : word;
   CExtWords : array[1..10] of word;
   CExtCnt,i : word;
begin
   ScanOperand(Dest, DExt, MSize);
   if (DExt.Mode=0) and (MSize <> L) then FlagError(807);
   if (DExt.Mode = 1) or ((DExt.Mode = 7) and (DExt.Reg = 4))
      then FlagError(806);
   InstrWord := $4200 + $08*DExt.Mode + DExt.Reg;
   case MSize of
      L :  InstrWord := InstrWord + $80;
      W :  InstrWord := InstrWord + $40;
   end;
   instr := 'clr.';
   case MSize of
      L :  instr := instr + T;
      W :  instr := instr + V;
      B :  instr := instr + 'b';
   end;
   while length(instr) < 8 do instr := instr + ' ';
   instr := instr + Dest;
   CExtCnt := 0;
   for i := 1 to DExt.ExtCnt do begin
      inc (CExtCnt);
      CExtWords[CExtCnt] := DExt.ExtWords[i];
   end;
   if 1+CExtCnt <= 4 then
      CListOut(instr, 1+CExtCnt,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3])
   else begin
      CListOut(instr,4,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3]);
      CListOut("",1+CExtCnt-4,CExtWords[4],CExtWords[5],CExtWords[6],0);
   end;
end;

procedure CCmp(MSize : SizeType;Source,Dest:string);
var SExt,DExt : ExtDescType;
   instr : string;
   InstrWord : word;
   CExtWords : array[1..10] of word;
   CExtCnt,i : word;
   UseTest : boolean;
begin
   ScanOperand(Source,SExt,MSize);
   ScanOperand(Dest,DExt,MSize);
   UseTest := false;
if ((SExt.Mode=7) and (SExt.Reg=4)) and (DExt.Mode <> 1) then begin
  if (DExt.Mode=7) and (DExt.Reg=4) then FlagError(805);
  if SExt.ImmVal = 0 then begin
    UseTest := true;
    instr := 'tst.4';
    InstrWord := $4a00 + $08*DExt.Mode + DExt.Reg;
    case MSize of
      L : InstrWord := InstrWord + $80;
      W : InstrWord := InstrWord + $40;
    end;
  end else begin
    instr := 'cmpi.';
    InstrWord := $0c00 + $08*DExt.Mode + DExt.Reg;
    case MSize of
      L : InstrWord := InstrWord + $80;
      W : InstrWord := InstrWord + $40;
    end;
  end;
end else begin
  if (DExt.Mode=0) then begin
    instr := 'cmp.4';
    case MSize of
      L : InstrWord := InstrWord + $80;
      W : InstrWord := InstrWord + $40;
    end;
  end else begin
    if DExt.Mode <> 1 then FlagError(805);
    instr := 'cmpa.4';
    InstrWord := $b0c0 + $200*DExt.Reg + $08*SExt.Mode + SExt.Reg;
    if MSize=L then InstrWord := InstrWord + $100;
  end;
end;

CExtCnt := 0;
if not UseTest then
  for i := 1 to SExt.ExtCnt do begin
    inc(CExtCnt);
    CExtWords[CExtCnt] := SExt.ExtWords[i];
  end;
for i := 1 to DExt.ExtCnt do begin
  inc(CExtCnt);
  CExtWords[CExtCnt] := DExt.ExtWords[i];
end;

case MSize of
  L : instr := instr + 'l';
  W : instr := instr + 'w';
  B : instr := instr + 'b';
end;
while length(instr) < 8 do instr := instr + ' ';

if not UseTest then 
  instr := instr + Source + ";
instr := instr + Dest;

if 1+CExtCnt <= 4 then
  CListOut(instr,1+CExtCnt,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3])
else begin
  CListOut(instr,4,InstrWord,CExtWords[1],CExtWords[2],CExtWords[3]);
  CListOut(C\' ,1+CExtCnt-4,CExtWords[4],CExtWords[5],CExtWords[6],0);
end;
end;

procedure DoBranch(Cond,LabI:longint);
const
    'blt','bgt','ble');
var LStr : string[6];
begin
  if Cond <> CondF then begin
    str(Labi,LStr);
    CListOut(BrInstr[Cond]+'.l L '+LStr,3,$60ff + $100*Cond,0,0,0);
    StoreLabelOffs(Labi);
  end;
end;

procedure PlaceLabelAct(Labl:longint);
var LStr : string[6];
begin
  str(Labi,LStr);
  ListOutC' +LStr+Y);
  DecLabelAddr (Labi);
  A3Valid := false;
end;

procedure PlaceForgetLabelAct(Labl:longint);
begin
  PlaceLabel(Labl);
  ForgetLabel(Labl);
end;

function GBTrunc(D:double):longint;
var Tmp : double;
begin
  Tmp := 1.0 * $40000000;
  Tmp := Tmp * 2;
  while D > Tmp do D := D - Tmp;
  Tmp := -Tmp;
  while D < Tmp do D := D - Tmp;
  GBTrunc := trunc(D);
function RealToLongint(R : single) : longint;
var
  L : longint absolute R;
begin
  RealToLongint := L;
end;

begin
  PlaceLabel := PlaceLabelAct;
  PlaceForgetLabel := PlaceForgetLabelAct;
end.
unit CircRef;

interface

uses Types;

type
  SimpleProc = procedure;

ScanVarProc = procedure(var I : integer; B1, B2 : boolean);

ScanParamProc = procedure(I1, I2 : integer; B : boolean);

ScanForProc = procedure(B : boolean);

ScanTypeProc = procedure(var I : integer);

ScanParmDecProc = procedure(var L : longint; B : boolean);

EmitProc = procedure(OC : OpCode; L : longint);

PlaceLabelProc = procedure(L : longint);

var
  ScanConstDec,
  ScanTypeDec,
  ScanProcDec,
  ScanFuncDec,
  ScanVarDec,
  ScanDeclarations,
  ScanParallelStat,
  ScanProcessCall,
  ScanStat,
  ScanCmpdStat : SimpleProc;

ScanVarAddress : ScanVarProc;

ScanParams : ScanParamProc;
ScanForStat : ScanForProc;

ScanType,
ScanProcessType,
ScanResourceType : ScanTypeProc;

ScanParmDec : ScanParmDecProc;

Emit : EmitProc;

PlaceLabel,
PlaceForgetLabel : PlaceLabelProc;

implementation

procedure DefaultSimpleProc;
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - Simple Proc');
  Halt(4);
end;

procedure DefaultScanVarProc(var I : integer; Bl, B2 : boolean);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanVar Proc');
  Halt(4);
end;

procedure DefaultScanParamProc(I1, I2 : integer; B : boolean);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanParam Proc');
  Halt(4);
end;

procedure DefaultScanForProc(B : boolean);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanFor Proc');
  Halt(4);
end;

procedure DefaultScanTypeProc(var I : integer);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanType Proc');
  Halt(4);
end;

procedure DefaultScanParmDecProc(var L : longint; B : boolean);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanParmDec Proc');
  Halt(4);
end;

procedure DefaultEmitProc(OC : OpCode; L : longint);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanEmit Proc');
  Halt(4);
end;

procedure DefaultPlaceLabelProc(L : longint);
begin
  writeln; writeln;
  writeln('Error - Uninitialized Procedural Variable - ScanPlace Proc');
  Halt(4);
end;

begin
  ScanConstDec := DefaultSimpleProc;
  ScanTypeDec := DefaultSimpleProc;
  ScanProcDec := DefaultSimpleProc;
  ScanFuncDec := DefaultSimpleProc;
  ScanVarDec := DefaultSimpleProc;
  ScanDeclarations := DefaultSimpleProc;
  ScanParallelStat := DefaultSimpleProc;
  ScanProcessCall := DefaultSimpleProc;
  ScanStat := DefaultSimpleProc;
  ScanCmpdStat := DefaultSimpleProc;
  ScanVarAddress := DefaultScanVarProc;
  ScanParams := DefaultScanParamProc;
  ScanForStat := DefaultScanForProc;
  ScanType := DefaultScanTypeProc;
  ScanProcessType := DefaultScanTypeProc;
  ScanResourceType := DefaultScanTypeProc;
  ScanParmDec := DefaultScanParmDecProc;
  Emit := DefaultEmitProc;
  PlaceLabel := DefaultPlaceLabelProc;
  PlaceForgetLabel := DefaultPlaceLabelProc;
end.
unit list;
uses crt, options, labels;

type lstring = string[78];

procedure ListOut(s:lstring);
procedure InitList;
procedure CloseList;
procedure CListOut(s:lstring; count:byte; w1, w2, w3, w4: word);

implementation

var f:text;
    wait: char;

procedure InitList;
begin
    assign(f,'zzzzz.tmp');
    rewrite(f);
end;

procedure CloseList;
begin
    close(f);
end;

procedure ListOut(s:lstring);
var C: char;
begin
    while pos(#13, s) <> 0 do s[pos(#13, s)] := ' ';
    if CheckOption('L') then C := #1 else C := #0;
    writeln(f,C+s);
end;

procedure CListOut(s:lstring; count:byte; w1, w2, w3, w4: word);
const
    TAB = #9;

var CStr: string[3];
var count: byte;

begin
    if CheckOption('Q') AND (s <> '') AND (count <> 0)
        AND (copy(s,1,3) <> 'dc.') then begin
        ListOut('1' + TAB + 'trap #0 ');
        StoreCodeWord($4e40);
    end;
    str(count, CStr);
    if s <> '' then ListOut(CStr+TAB+s)
    else ListOut(CStr);
    if count > 4 then count := 4;
    if count >= 1 then StoreCodeWord(w1);
    if count >= 2 then StoreCodeWord(w2);
if count >= 3 then StoreCodeWord(w3);
if count >= 4 then StoreCodeWord(w4);
end;

end.

unit labels;
{used to generate internal labels and storage areas}

interface
uses errors,tables,dos;

type
  SStr = string[15];
  LineStr = string[81];

procedure InitLabels;
procedure DiscardLabels;
function NewLabel:longint;
procedure DecLabelAddr(LabelNum:longint);
procedure ForgetLabel(LabelNum:longint);
procedure StoreLabelOffs(LabelNum:longint);
procedure StoreCodeWord(CodeWord:word);
procedure LocateErrorSetup;
function HexStrL(value,width:longint):SStr;

var
  CurrentAddress : longint;

implementation

const
  MaxNumLabels = 512;
  MaxNumFwdRef = 512;

  PointerRecordType = array[1..MaxNumLabels] of
    record
      LabelNum,
      address : longint;
    end;

  ForwardRefType = array[1..MaxNumFwdRef] of
    record
      LabelNum,
      offset : longint;
    end;

  PByte = ^byte;

  ObjType = record
    Address,Len : longint;
    CodeBytes : array[0..1023] of byte;
  end;

  ObjType;
StubDataDesc = record
  Identifier : string[15];
  LoadExeName : string[64];
  FileOffset : longint;
end;

const
  CodeCachePageSize = 4096;
  CodeCacheOffsetMask = CodeCachePageSize - 1;
  CodeCachePageMask = $fffffff - CodeCacheOffsetMask;
  CodeCachePageCount = 4;

type
  CodeCachePageDesc = record
    CodeData : array[0..CodeCacheOffsetMask] of byte;
    CodeAddr : longint;
    PageDirty : boolean;
  end;

CodeCachePagesDesc = array[1..CodeCachePageCount] of CodeCachePageDesc;

var
  CurrentLabel: longint;
  PointerRecord : ~ PointerRecordType;
  ForwardRef: ~ ForwardRefType;
  NumCurrentLabels,
  NumFwdRef: word;
  DoLocateError: boolean;
  LocateErrAddr,LocateErrNum : longint;
  PCodeCache : ^CodeCachePagesDesc;
  CacheFileCreated : boolean;
  CacheFile : file of CodeCachePageDesc;
  CacheFileSize,
  CachePagesCreated : longint;
  OldExitProc : pointer;
  ReadCount,
  WriteCount,
  MissCount,
  HitCount,
  ShiftCount : longint;

procedure SaveCodeCachePage(var P:CodeCachePageDesc);
var
  Temp : CodeCachePageDesc;
  Found : boolean;
  i : longint;
begin
  inc(WriteCount);
  if not CacheFileCreated then begin
procedure LoadCodeCachePage(PageAddr: longint; var P: CodeCachePageDesc);
var
  Found: boolean;
  i: longint;
begin
  inc(ReadCount);
  if not CacheFileCreated then FlagError(2222);
  i := 0; Found := false;
  while (i < CacheFileSize) AND not Found do begin
    seek(CacheFile, i);
    read(CacheFile, Temp);
    if Temp.CodeAddr = P.CodeAddr then Found := true
    else inc(i);
  end;
  if not Found then inc(CacheFileSize);
  seek(CacheFile, i);
  write(CacheFile, P);
end;

procedure FindCodePage(PageAddr: longint);
var
  ij: byte;
  Temp: CodeCachePageDesc;
begin
  i := 1;
  { See if the page is present in the cache. }
  while (i <= CodeCachePageCount) AND (PCodeCache^[i].CodeAddr <> PageAddr)
    do inc(i);
  if i <= CodeCachePageCount then begin
    { Page is Present - shift it to first position, others up by one }
    inc(ShiftCount);
    inc(HitCount);
    Temp := PCodeCache^[i];
    for j := i downto 2 do
      PCodeCache^[j] := PCodeCache^[j-1];
PCodeCache^[1] := Temp;
Exit;
end;
if PageAddr > (CachePagesCreated * CodeCachePageSize • 1) then begin
  { Page is new - Save least recently used page if necessary }
  if PCodeCache^[CodeCachePageCount].PageDirty then
    SaveCodeCachePage(PCodeCache^[CodeCachePageCount]);
  for j := CodeCachePageCount downto 2 do
    PCodeCache^[j] := PCodeCache^[j-1];
  FillChar(PCodeCache^[1],SizeOf(PCodeCache^[1]),0);
  with PCodeCache^[1] do begin
    CodeAddr := PageAddr;
    PageDirty := false;
  end;
  inc(CachePagesCreated);
  Exit;
end;
{ Page is not new, but it isn't in memory either. }
inc(MissCount);
{ Get rid of least recently used page. }
if PCodeCache^[CodeCachePageCount].PageDirty then
  SaveCodeCachePage(PCodeCache^[CodeCachePageCount]);
{ Shift other pages down one in position. }
for j := CodeCachePageCount downto 2 do
  PCodeCache^[j] := PCodeCache^[j-1];
{ Read in desired code page. }
LoadCodeCachePage(PageAddr,PCodeCache^[1]);
PCodeCache^[1].PageDirty := false;
end;

function Code(CodeAddr:longint):PByte;
begin
  if PCodeCache^[1].CodeAddr <> (CodeAddr AND CodeCachePageMask) then
    FindCodePage(CodeAddr AND CodeCachePageMask)
  else inc(HitCount);
  Code := @PCodeCache^[1].CodeData[CodeAddr AND CodeCacheOffsetMask];
end;

procedure MarkDirtyCode;
begin
  PCodeCache^[1].PageDirty := true;
end;

function NewLabel:longint;
begin
  inc(CurrentLabel);
  NewLabel := CurrentLabel;
end;

procedure LocateErrorSetup;
var ErrFile : text;
begin
assign(ErrFile,'error.###');
{$I-}reset(ErrFile);{$I+}
if IoResult = 0 then begin
  DoLocateError := true;
  readln(ErrFile,LocateErrAddr);
  readln(ErrFile,LocateErrNum);
  close(ErrFile);
  erase(ErrFile);
end;
end;

procedure InitLabels;
var
  i : byte;
begin
  CurrentLabel := 0;
  NumCurrentLabels := 0;
  DoLocateError := false;
  new(PointerRecord);
  new(ForwardRef);
  new(CodeCache);
  for i := 1 to CodeCachePageCount do with CodeCache^ [i] do begin
    CodeAddr := -1;
    PageDirty := false;
  end;
  CacheFileCreated := false;
  assign(CacheFile,GetEnv('TEMP') + '\BPLSPILL.$$');
  CachePagesCreated := 0;
  CacheFileSize := 0;
  ReadCount := 0;
  WriteCount := 0;
  MissCount := 0;
  HitCount := 0;
  ShiftCount := 0;
  CurrentAddress := 0;
  NumFwdRef := 0;
end;

function Extended(S:LineStr;W:byte):LineStr;
begin
  while Length(S) < W do S := S + ' ';
  Extended := copy(S,W,W);
end;

function HexStrL(value,width:longint):SStr;
const HexChars : array[0..15] of char =
  ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
var tmp:SStr;
begin
tmp := ";
while width > 0 do begin
    tmp := HexChars[value and $0f] + tmp;
    dec(width);
    value := value shr 4;
end;
HexStrL := tmp;
end;

function ByteHexStr(FirstByte:longint;Count:byte):LineStr;
var
    Tmp : LineStr;
begin
    Tmp := ";
    while Count > 0 do begin
        Tmp := Tmp + HexStrL(code(FirstByte)^,2);
        inc(FirstByte); dec(Count);
    end;
    ByteHexStr := Tmp;
end;

function SRecStr(stype:byte;Address:longint;cnt:word;CodePtr:longint):string;
var s:string;
    Chk :  longint;
    i:word;
    b :  byte;
begin
    s := 'S' + chr(stype + ord('0')) + HexStrL(cnt+5,2) + HexStrL(Address,8);
    Chk := cnt + 5;
    Chk := chk + ((Address shr 24) and $ff) + ((Address shr 16) and $ff) + ((Address shr 8) and $ff) + (Address and $ff);
    if cnt <> 0 then for i := 0 to cnt-1 do begin
        b := Code(CodePtr+i)^;
        chk := chk + b;
        s := s + HexStrL(b,2);
    end;
    chk := chk and $ff;
    chk := 255 - chk;
    SRecStr := s + HexStrL(chk,2);
end;

procedure MoveStubToObj(var ObjFile :file);
var
    StubFile : file;
    StubFileSize : longint;
    StubData : pointer;
    Dir : DirStr;
    Name : NameStr;
    Ext : ExtStr;
    i : word;
    P : ^StubDataDesc;
Found : boolean;
begin
  FSplit(ParamStr(0),Dir,Name,Ext);
  assign(StubFile,Dir + 'bplstub.stb');
  reset(StubFile,1);
  StubFileSize := FileSize(StubFile);
  GetMem(StubData,StubFileSize);
  BlockRead(StubFile,StubData ^ ,StubFileSize);
  close(StubFile);
  i := 0;
  Found := false;
  while not Found AND (i < StubFileSize) do begin
    P := ptr(seg(StubData ^ ),ofs(StubData ^ ) + i);
    if P ^ .Identifier = 'nnamgreB gerG' then Found := true
    else inc(i);
  end;
  if Found then begin
    P ^ .FileOffset := StubFileSize;
    BlockWrite(ObjFile,StubData ^ ,StubFileSize);
  end;
  FreeMem(StubData,StubFileSize);
end;

procedure DiscardLabels;
var
  Fin : text;
  FOut : file;
  CodePtr : longint;
  InString : string[255];
  LLength : longint;
  ChkByte : longint;
  BinFile : file of byte;
  i : integer;
  OutBuffer : array[1..4096] of char;
  OutBufferPtr : word;
  ObjFile : file;
 _ObjData : ObjType;
  DoLst,
  DoObj : boolean;
  C : char;

procedure FlushBuffer;
begin
  if OutBufferPtr <> 1 then begin
    BlockWrite(FOut,OutBuffer,OutBufferPtr-1);
    OutBufferPtr := 1;
  end;
end;

procedure TextOut(s:string);
var i:integer;
begin
  if (OutBufferPtr + Length(s)) >= 4096 then FlushBuffer;
  for i := 1 to length(s) do begin
    OutBuffer[OutBufferPtr] := s[i];
    inc(OutBufferPtr);
  end;
end;

begin
  DoObj := true;
  for i := 2 to ParamCount do
    if ParamStr(i) = '/s' then DoObj := false;
  if not DoObj then begin
    assign(FOut,MainFileName+'.run');
    rewrite(FOut,1);
    OutBufferPtr := 1;
    CodePtr := 0;
    while CodePtr < CurrentAddress do begin
      LLength := 16;
      if CurrentAddress-CodePtr < 16 then LLength := CurrentAddress-CodePtr;
      TextOut(SrecStr(3,RomCodeBase+CodePtr,LLength,CodePtr)+#$0d+#$0a);
      inc(CodePtr,16);
    end;
    TextOut(SrecStr(9,RomCodeBase,0,0)+#$0d+#$0a);
    FlushBuffer;
    close(FOut);
  end else begin
    assign(ObjFile,MainFileName+'.exe');
    rewrite(ObjFile,1);
    MoveStubToObj(ObjFile);
    CodePtr := 0;
    while CodePtr < CurrentAddress do begin
      ObjData.Address := RomCodeBase+CodePtr;
      ObjData.Len := 1024;
      if CurrentAddress-CodePtr < 1024 then
        ObjData.Len := CurrentAddress-CodePtr;
      for i := 0 to ObjData.Len-1 do
        ObjData.CodeBytes[i] := code(CodePtr+i) ~ ;
      BlockWrite(ObjFile,ObjData,SizeOf(ObjData));
      CodePtr := CodePtr + ObjData.Len;
    end;
    close(ObjFile);
  end;
end;

DoLst := false;
for i := 2 to ParamCount do
  if ParamStr(i) = '/l' then DoLst := true;
if DoLst then begin
  assign(FIn,'zzzzzz.tmp');
  reset(FIn);
  assign(FOut,MainFileName+'.lst');
rewrite (FOut, 1);
OutBufferPtr := 1;
CodePtr := 0;
while not eof(FIn) do
begin
  readln (FIn, Instring);
  C := Instring[1];
  InString := copy(InString, 2, 255);
end;

case InString[1] of
  '*': if C = #1 then TextOut('******' + InString + #$0d + #$0a);
  'L': if C = #1 then TextOut(' ' + InString + #$0d + #$0a);
  'O': if C = #1 then TextOut('0' + copy(InString, 2, 255) + #$0d + #$0a);
  '1': begin
    if C = #1 then begin
      if length(InString) > 1 then
        TextOut(Extended(HexStrL(CodePtr, 6), 7)
          + Extended(ByteHexStr(CodePtr, 2), 9)
          + copy(InString, 2, 255) + #$0d + #$0a)
      else
        TextOut(Extended(',', 7)
          + Extended(ByteHexStr(CodePtr, 2), 9)
          + copy(InString, 2, 255) + #$0d + #$0a);
    end;
    inc(CodePtr, 2);
  end;
  '2': begin
    if C = #1 then begin
      if length(InString) > 1 then
        TextOut(Extended(HexStrL(CodePtr, 6), 7)
          + Extended(ByteHexStr(CodePtr, 2), 9)
          + copy(InString, 2, 255) + #$0d + #$0a)
      else
        TextOut(Extended(',', 7)
          + Extended(ByteHexStr(CodePtr, 2), 9)
          + copy(InString, 2, 255) + #$0d + #$0a);
    end;
    inc(CodePtr, 4);
  end;
  '3': begin
    if C = #1 then begin
      if length(InString) > 1 then
        TextOut(Extended(HexStrL(CodePtr, 6), 7)
          + Extended(ByteHexStr(CodePtr, 4), 9)
          + copy(InString, 2, 255) + #$0d + #$0a)
      else
        TextOut(Extended(',', 7)
          + Extended(ByteHexStr(CodePtr, 4), 9)
          + copy(InString, 2, 255) + #$0d + #$0a);
    end;
    inc(CodePtr, 4);
if $C = \#1$ then TextOut(Extended(","7) + ByteHexStr(CodePtr,2) + #$0d + #$0a);

inc(CodePtr,2);
end;

'4', '5', '6', '7': begin
  if $C = \#1$ then begin
    if Length(InString) > 1 then
      TextOut(Extended(HexStrL(CodePtr,6),7)
        + Extended(ByteHexStr(CodePtr,4),9)
        + copy(InString,2,255)+#$0d+$#0a)
    else
      TextOut(Extended(","7)
        + Extended(ByteHexStr(CodePtr,4),9)
        + copy(InString,2,255)+#$0d+$#0a);

    inc(CodePtr,4);
    if $C = \#1$ then
      TextOut(Extended(","7)
        + Extended(ByteHexStr(CodePtr,4),9)
        + #$0d + #$0a);
    inc(CodePtr,4);
  end;
  else if $C = \#1$ then
    TextOut("'");
    inc(CodePtr,4);
end;

close(FIn);
FlushBuffer;
close(FOut);
end;
assign(FIn,'zzzzz.tmp');
erase(FIn);
dispose(PointerRecord);
dispose(ForwardRef);
dispose(PCodeCache);
if CacheFileCreated then close(CacheFile);
if NumFwdRef <> 0 then FlagError(-7);
end;

{F+} procedure LabelExitProc;
begin
  if IOResult <> 0 then ;
  {$I-} ExitProc := OldExitProc; {$I+}
  if IOResult <> 0 then ;
  if CacheFileCreated then erase(CacheFile);
end;

procedure DecLabelAddr(LabelNum:longint);
var i : word;
  found : boolean;
addr,
offset : longint;
begin
inc(NumCurrentLabels);
if NumCurrentLabels > MaxNumLabels then FlagError(275);
PointerRecord^[NumCurrentLabels].LabelNum := LabelNum;
PointerRecord^[NumCurrentLabels].address := CurrentAddress;
repeat
found := false;
i := 1;
while (i <= NumFwdRef) and (not found) do
  if ForwardRef^[i].LabelNum = LabelNum then found := true
  else inc(i);
if found then begin
  offset := CurrentAddress - ForwardRef^[i].offset;
  addr := ForwardRef^[i].offset;
  code(addr) := ((offset and $ff000000) div $1000000) and $ff;
  MarkDirtyCode;
  code(addr+1) := ((offset and $ff0000) div $10000) and $ff;
  MarkDirtyCode;
  code(addr+2) := ((offset and $ff00) div $100) and $ff;
  MarkDirtyCode;
  code(addr+3) := offset and $ff;
  MarkDirtyCode;
  ForwardRef^[i] := ForwardRef^[NumFwdRef];
  dec(NumFwdRef);
end;
until not found;
end;

procedure ForgetLabel(LabelNum:longint);
var i : word;
begin
i := 1;
while (i <= NumCurrentLabels) and
  (PointerRecord^[i].LabelNum <> LabelNum) do inc(i);
if i > NumCurrentLabels then FlagError(-6);
PointerRecord^[i] := PointerRecord^[NumCurrentLabels];
dec(NumCurrentLabels);
end;

procedure StoreLabelOffs(LabelNum:longint);
var i:word;
addr:longint;
begin
i := 1;
while (i <= NumCurrentLabels) and
  (PointerRecord^[i].LabelNum <> LabelNum) do inc(i);
if i <= NumCurrentLabels then begin
  addr := CurrentAddress - 4;
  addr := PointerRecord^[i].address - addr;
begin
OldExitProc := ExitProc;
ExitProc := @LabelExitProc;
end.
unit tables;
{stores all compiler created tables}

interface
uses crt,types,errors;

const
MaxNumVars = 300;
MaxNumId = 600;
MaxNumType = 90;
MaxNumEnum = 50;
MaxNumFields = 80;
MaxNumProcs = 150;
MaxNumFuncs = 50;
MaxNumParams = 350;
MaxNumEnums = 16;

\begin{verbatim}

\text{code(CurrentAddress-4)} := ((\text{addr and $ff000000$} \ \text{div $1000000$}) \ \text{and $ff$});
\text{MarkDirtyCode};
\text{code(CurrentAddress-3)} := ((\text{addr and $ff0000$} \ \text{div $10000$}) \ \text{and $ff$});
\text{MarkDirtyCode};
\text{code(CurrentAddress-2)} := ((\text{addr and $ff00$} \ \text{div $100$}) \ \text{and $ff$});
\text{MarkDirtyCode};
\text{code(CurrentAddress-1)} := \text{addr and $ff$};
\text{MarkDirtyCode};
end\ else \ begin 
\text{inc(NumFwdRef)};
\text{if NumFwdRef} > \text{MaxNumFwdRef} \ \text{then FlagError(277)};
\text{ForwardRef}^[\text{NumFwdRef}].\text{LabelNum} := \text{LabelNum};
\text{ForwardRef}^[\text{NumFwdRef}].\text{offset} := \text{CurrentAddress-4};
\end;
\end;

\text{procedure StoreCodeWord(CodeWord:word);}
begin
\text{if DoLocateError} \ \text{then begin}
\text{if (LocateErrAddr} \ \leq \ \text{CurrentAddress + RomCodeBase} \ \text{and}
(\text{CurrentAddress} \ \text{<} \ 0) \ \text{then}
\text{FlagError(LocateErrNum)};
\end;
\text{code(CurrentAddress)} := \text{CodeWord div $100$} \ \text{AND $ff$}; \ \text{hi byte first}
\text{MarkDirtyCode};
\text{inc(CurrentAddress)};
\text{code(CurrentAddress)} := \text{CodeWord AND $ff$}; \ \text{lo byte second}
\text{MarkDirtyCode};
\text{inc(CurrentAddress)};
\end;

begin
\text{OldExitProc} := \text{ExitProc};
\text{ExitProc} := @\text{LabelExitProc};
\end.
end.
MaxNumFiles = 20;
MaxNestedProcess = 10;
MaxNestedExit = 30;

type
  IdTablType = array[0..MaxNumId] of IdRec;
  TypeTablType = array[0..MaxNumType] of TypeDataType;
  EnumTablType = array[0..MaxNumEnum] of longint;
  FieldTablType = array[0..MaxNumFields] of FieldDescType;
  VarTablType = array[0..MaxNumVars] of VarDataType;
  ProcTablType = array[0..MaxNumProcs] of ProcEntryType;
  FuncTablType = array[0..MaxNumFuncs] of FuncEntryType;
  ParamTablType = array[0..MaxNumParams] of ParamEntryType;
  FileTablType = array[0..MaxNumFiles] of string[64];
  ExitTablType = array[0..MaxNestedExit] of longint;
  StackTablType = array[0..MaxNestedExit] of longint;

var
  IdTabl : ^IdTablType;
  TypeTabl : ^TypeTablType;
  EnumTabl : ^EnumTablType;
  FieldTabl : ^FieldTablType;
  VarTabl : ^VarTablType;
  ProcTabl : ^ProcTablType;
  FuncTabl : ^FuncTablType;
  ParamTabl : ^ParamTablType;
  FileTabl : ^FileTablType;
  ExitTabl : ^ExitTablType;
  StackTabl : ^StackTablType;
  IdTablPtr,
  TypeTablPtr,
  EnumTablPtr,
  FieldTablPtr,
  VarTablPtr,
  ProcTablPtr,
  FuncTablPtr,
  ParamTablPtr,
  FileTablPtr,
  ExitTablPtr,
  StackTablPtr,
  BlockLevel : integer;
  ProcessLevel : array[1..MaxNestedProcess] of integer;
  ProcessLevelPtr : integer;
  IntTypePtr,
  LongintTypePtr,
  ShortintTypePtr,
  ByteTypePtr,
  WordTypePtr,
  RealTypePtr,
  DoubleTypePtr,
  BoolTypePtr,
procedure InitTables;
    procedure SelectRomCode(VarBase, CodeBase: longint);
    procedure DiscardTables;
    procedure DumpTables;
    procedure StoreTypeData(TypeD: TypeDescType; sub: longint;
                              i1, i2: longint; size: longint; mask: longint);
    procedure StoreVarData(TypePtr, AddrOffs: longint);
    function AlreadyDef(id: longint): boolean;
    procedure DecConst(ctyp: IdDesc; id, value: longint);
    procedure DecType(id, TypePtr: longint);
    function ContainsType(TypeD: TypeDescType; TypePtr, nest: longint): boolean;
    procedure DeclareVar(id, VarPtr: integer);
    procedure DecRproc(id, VarPtr: integer);
    procedure DecProc(id, ProcPtr: integer);
    procedure DecFunc(id, FuncPtr: integer);
    procedure DecParam(id, VarPtr: integer; ParmDesc: ParmType);
    procedure GetIdDesc(IdNmbr: integer; var result: IdRec; var Found: boolean);
    procedure MarkForwardDec(fwdrow, fwdcol, fwdfnum: integer);
    procedure StoreParamData(TypePtr: integer; address: longint; valvar: ParmType;
                              idNumber: integer);
    procedure StoreFuncData(flabel: longint; fparamptr: integer);
    procedure StoreProcData(plabel: longint; fparamptr: integer);
    procedure AdjustAddr(var address: longint; mask: longint);
    procedure StoreFileName(FileName: DataString; var NameNum: integer);
    procedure GetFileName(var FileName: DataString; NameNum: integer);
    procedure StartProcessLevel;
    procedure EndProcessLevel;
    function ArrayDepth (p: integer): longint;
    function StackSize: longint;
    procedure AdjustStack(Size: longint);
    function ExitLabel: longint;
    procedure NewExitLevel(ExitLabel: longint);
    procedure EndExitLevel;

implementation

procedure IncTablPtr(var TablPtr: integer; limit: longint; ErrorNum: integer);
begin
    inc(TablPtr);
    {See if the table pointer has exceeded its maximum value}
{If so, flag the error and halt.}
if TblPtr > limit then FlagError(ErrorNum);
end;

procedure DumpTabs;
begin
end;

procedure AdjustAddr(var address:longint;mask:longint);
begi
    address := (address + mask) AND (not mask);
end;

procedure StoreTypeData(TypeD:TypeDescType;sub:longint;
    i1,i2:longint;size:longint;mask:longint);
begin
    {Increment the type table pointer, checking for overflow.}
    IncTablPtr(TypeTablPtr,MaxNumType,262);
    TypeTabl^ [TypeTablPtr].TypeDesc := TypeD;
    TypeTabl^ [TypeTablPtr].SubType := sub;
    TypeTabl^ [TypeTablPtr].p1 := i1;
    TypeTabl^ [TypeTablPtr].p2 := i2;
    TypeTabl^ [TypeTablPtr].TypeSize := size;
    TypeTabl^ [TypeTablPtr].AddrMask := mask;
    { Indicate the current block level.}
    TypeTabl^ [TypeTablPtr].level := BlockLevel;
end;

function ContainsType(TypeD:TypeDescType;TypePtr,nest:longint):boolean;
var tmp, done:boolean; i,tt:longint;
begin
    tmp := false; done := false;
    if nest > 25 then done := true;
    while not done and not tmp do begin
        if TypeTabl^ [TypePtr].TypeDesc=TypeD then tmp := true
        else begin
            case TypeTabl^ [TypePtr].TypeDesc of
                ArrayType : begin
                    TypePtr := TypeTabl^ [TypePtr].p2;
                    inc(nest);
                end;
                RecordType : begin
                    i := TypeTabl^ [TypePtr].p1;
                    while not tmp and (i < TypeTabl^ [TypePtr].p1 +
                        TypeTabl^ [TypePtr].p2) do begin
                        tt := i;
                        tmp := ContainsType(TypeD,
                            FieldTabl^ [i].FTypePtr,nest+1);
                        i := i + 1;
                    end;
                end;
            end;
        end;
    end;
end;
end;
    done := true;
end;

ChannelType,
SharedType,
PtrType: begin
    TypePtr := TypeTabl^[TypePtr].SubType;
    inc(nest);
end;
else done := true;
end;
end;
end;
ContainsType := tmp;
end;

procedure StoreVarData(TypePtr,AddrOffs:longint);
begin
    {Increment the var table pointer, checking for overflow.}
    IncTablPtr(VarTablPtr,MaxNumVars,265);
    {Store the various parameters associated with variables.}
    with VarTabl^[VarTablPtr] do begin
        VarTypePtr := TypePtr;
        AddrOffset := AddrOffs;
        {Indicate the current block level.}
        level := BlockLevel;
        IsAbsolute := NotAbs;
        AbsOffset := 0;
        AbsLevel := -1;
    end;
end;

procedure AddStandardTypes;
begin
    StoreTypeData(DoubleType,0,0,0,8,1);
    DoubleTypePtr := TypeTablPtr;
    StoreTypeData(RealType,0,0,0,4,1);
    RealTypePtr := TypeTablPtr;
    StoreTypeData(LongintType,0,-2,2147483647,2147483647,4,1);
    LongintTypePtr := TypeTablPtr;
    StoreTypeData(IntType,0,-32768,32767,2,1);
    IntTypePtr := TypeTablPtr;
    StoreTypeData(WordType,0,0,65535,2,1);
    WordTypePtr := TypeTablPtr;
    StoreTypeData(ShortintType,0,-128,127,1,0);
    ShortintTypePtr := TypeTablPtr;
    StoreTypeData(ByteType,0,0,255,1,0);
    ByteTypePtr := TypeTablPtr;
    StoreTypeData(BoolType,0,0,1,1,0);
    BoolTypePtr := TypeTablPtr;
    StoreTypeData(CharType,0,0,255,1,0);
CharTypePtr := TypeTablPtr;
StoreTypeData(TextType,0,0,255,90,1);
TextTypePtr := TypeTablPtr;
StoreTypeData(StringType,0,255,0,256,0);
StringTypePtr := TypeTablPtr;
StoreTypeData(PtrType,0,0,0,4,3);
NilPointerTypePtr := TypeTablPtr;
end;

procedure SelectRomCode(VarBase,CodeBase:longint);
begin
  if RomCode then FlagError(279);
  if BlockLevel <> 0 then FlagError(279);
  if VarTablPtr <> 0 then FlagError(279);
  if (RomCodeBase and 3) <> 0 then FlagError(296);
  RomCode := true;
  RomVarBase := VarBase;
  RomCodeBase := CodeBase;
end;

procedure InitTables;
begin
  RomCode := false;
  RomVarBase := 0;
  RomCodeBase := $e0000000; {default code address}
  VarTablPtr := 0;
  newCVarTabl);
  FieldTablPtr := 0; {no field names}
  new(FieldTabl);
  EnumTablPtr := 0; {no enumerated identifiers}
  new(EnumTabl);
  IdTablPtr := 0; {no identifiers stored yet}
  new(IdTabl);
  TypeTablPtr := 0; {no types yet either}
  new(TypeTabl);
  ProcTablPtr := 0;
  new(ProcTabl);
  FuncTablPtr := 0;
  new(FuncTabl);
  ParamTablPtr := 0;
  new(ParamTabl);
  FileTablPtr := 0;
  new(FileTabl);
  ExitTablPtr := 0;
  new(ExitTabl);
  StackTablPtr := 0;
  new(StackTabl);
  AddStandardTypes;
  ProcessLevelPtr := 0;
  NumRequests := 10;
end;
procedure DiscardTables;
begin
  dispose(ParamTabl);
  dispose(FuncTabl);
  dispose(ProcTabl);
  dispose(TypeTabl);
  dispose(IdTabl);
  dispose(EnumTabl);
  dispose(FieldTabl);
  dispose(VarTabl);
end;

function AlreadyDef(id:longint):boolean;
var i : longint; tmp : boolean;
begin
  {start searching for the id at the top of push down stack}
  i := IdTablPtr;
  {Assume the id is not already defined - if we find it, make this true.}
  tmp := false;
  while (i<>0) and (not tmp) and (IdTabl^[i].level = BlockLevel) do begin
    if IdTabl^[i].IdNum = id then tmp := true;
    dec(i);
  end;
  AlreadyDef := tmp;
end;

procedure DecConst(ctyp:IdDesc;id,cvalue:longint);
var s:DataString;
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumId,261);
  {indicate the name of the constant}
  IdTabl^[IdTablPtr].IdNum := id;
  {indicate the value of the constant}
  IdTabl^[IdTablPtr].valptr := cvalue;
  {Indicate the current block level.}
  IdTabl^[IdTablPtr].level := BlockLevel;
  {Indicate that the identifier is a constant name.}
  IdTabl^[IdTablPtr].IdTyp := ctyp;
end;

procedure DecType(id,TypePtr:longint);
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumId,261);
  {indicate the name of the type}
  IdTabl^[IdTablPtr].IdNum := id;
  {pointer to the type data}
  IdTabl^[IdTablPtr].valptr := TypePtr;
  {Indicate the current block level.}
  IdTabl^[IdTablPtr].level := BlockLevel;
procedure DeclareVar(id,VarPtr:integer);
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumld,261);
  {indicate the name of the var}
  IdTabl^[IdTablPtr].IdNum := id;
  {pointer to the var data}
  IdTabl^[IdTablPtr].valptr := VarPtr;
  { Indicate the current block level.}
  IdTabl^[IdTablPtr].level := BlockLevel;
  {Indicate that the identifier is a variable name.}
  IdTabl^[IdTablPtr].IdTyp := VarName;
end;

procedure DecRproc(id,VarPtr:integer);
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumld,261);
  {indicate the name of the resource procedure}
  IdTabl^[IdTablPtr].IdNum := id;
  {pointer to the proc's data}
  IdTabl^[IdTablPtr].valptr := VarPtr;
  { Indicate the current block level.}
  IdTabl^[IdTablPtr].level := BlockLevel;
  {Indicate that the identifier is a resource procedure name.}
  IdTabl^[IdTablPtr].IdTyp := DecRsrcProc;
end;

procedure DecProc(id,ProcPtr:integer);
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumld,261);
  {indicate the name of the procedure}
  IdTabl^[IdTablPtr].IdNum := id;
  {pointer to the data for the procedure}
  IdTabl^[IdTablPtr].valptr := ProcPtr;
  { Indicate the current block level.}
  IdTabl^[IdTablPtr].level := BlockLevel;
  {Indicate that the identifier is a procedure name.}
  IdTabl^[IdTablPtr].IdTyp := ProcName;
end;

procedure DecFunc(id,FuncPtr:integer);
begin
  {Increment the ID table pointer, checking for overflow.}
  IncTablPtr(IdTablPtr,MaxNumld,261);
  {indicate the name of the function}
IdTabl^[IdTablPtr].IdNum := id;
{pointer to the data for the function}
IdTabl^[IdTablPtr].valptr := FuncPtr;
{Indicate the current block level.}
IdTabl^[IdTablPtr].level := BlockLevel;
{Indicate that the identifier is a function name.}
IdTabl^[IdTablPtr].IdTyp := FuncName;
end;

procedure DecParam(id,VarPtr:integer;ParmDesc : ParmType);
begin
{Increment the ID table pointer, checking for overflow.}
IncTablPtr(IdTablPtr,MaxNumId,261);
{indicate the name of the parameter}
IdTabl^[IdTablPtr].IdNum := id;
{pointer to the data for the parameter}
IdTabl^[IdTablPtr].valptr := VarPtr;
{Indicate the current block level.}
IdTabl^[IdTablPtr].level := BlockLevel;
{Indicate that the identifier is a val, var, or ret parameter name.}
case ParmDesc of
  VarPtype : IdTabl^[IdTablPtr].IdTyp := VarParamName;
  ValPtype : IdTabl^[IdTablPtr].IdTyp := ValParamName;
  RetPtype : IdTabl^[IdTablPtr].IdTyp := RetParamName;
end;
end;

procedure MarkForwardDec(fwdrow,fwdcol,fwdfnum:integer);
var i:integer;
begin
  i := IdTablPtr;
  repeat
    if (i <> 0) then
      case IdTabl^[i].IdTyp of
        ProcName : begin
          {Indicate that the identifier is a fwd proc name.}
          IdTabl^[i].IdTyp := FwdProcName;
          with ProcTabl^[IdTabl^[i].valptr] do begin
            row := fwdrow;
            col := fwdcol;
            fnum := fwdfnum;
          end;
        end;
        FuncName : begin
          {Indicate that the identifier is a fwd func name.}
          IdTabl^[i].IdTyp := FwdFuncName;
          with FuncTabl^[IdTabl^[i].valptr] do begin
            row := fwdrow;
            col := fwdcol;
            fnum := fwdfnum;
          end;
        end;
  end;
end;
end;
else dec(i);
end;
until (i=0) OR (IdTabl^[i].IdTyp in [FwdProcName,FwdFuncName]);
end;

procedure GetIdDesc(IdNmbr:integer;var result:IdRec;var Found:boolean);
var idongint;
begin
{Not Found yet - set to true if and when Found}
Found := false;
{Start looking at top of stack, working down until Found or at end of stack}
i := IdTablPtr;
while (i>0) and (not Found) do
if IdTabl^[i].IdNum = IdNmbr then begin
Found := true;
{check for variables, etc out of process}
if ProcessLevelPtr <> 0 then begin
if (IdTabl^[i].level < ProcessLevel[ProcessLevelPtr]) and
not (IdTabl^[i].IdTyp in [LongintConst,CharConst,StrConst,
BoolConst,EnumConst,TypeName,RealConst]) then begin
Found := false;
dec(i);
end;
end;
end;
end
{Flag Found if successful, otherwise keep looking}
else dec(i);
{Return the result if successful}
if Found then result := IdTabl^[i];
end;

procedure StoreProcData(plabel:longint;fparamptr:integer);
begin
{Increment the procedure table pointer, checking for overflow.}
IncTablPtr(ProcTablPtr,MaxNumProcs,269);
{Store the various parameters associated with procedures.}
with ProcTabl^[ProcTablPtr] do begin
CodeLabel := plabel;
FirstParam := fparamptr;
{Indicate the current block level.}
level := BlockLevel;
end;
end;

procedure StoreFuncData(flabel:longint;fparamptr:integer);
begin
{Increment the function table pointer, checking for overflow.}
IncTablPtr(FuncTablPtr,MaxNumFuncs,270);
{Store the various parameters associated with functions.}
with FuncTabl^[FuncTablPtr] do begin
CodeLabel := flabel;
FirstParam := fparamptr;
{ Indicate the current block level.}
level := BlockLevel;
end;
end;

procedure StoreParamData(TypePtr:integer;address:longint;valvar:ParmTyp;
   IdNumber:integer);
begind
  {Increment the parameter table pointer, checking for overflow.}
  IncTablPtr(ParamTablPtr,MaxNumParams,271);
  {Store the various parameters associated with parameters.}
  with ParamTabl^ [ParamTablPtr] do begin
    IsVar := valvar;
    TypePointer := TypePtr;
    AddrOffs := address;
    IdNum := IdNumber;
    IsProcParm := false;
  end;
end;

procedure StoreFileName(FileName:DataString;var NameNum:integer);
begind
  IncTablPtr(FileTablPtr,MaxNumFiles,274);
  FileTabl^ [FileTablPtr] := FileName;
  NameNum := FileTablPtr;
end;

procedure GetFileName(var FileName:DataString;NameNum:integer);
begind
  FileName := FileTabl^ [NameNum];
end;

procedure StartProcessLevel;
begind
  inc(ProcessLevelPtr);
  if ProcessLevelPtr > MaxNestedProcess then FlagError(287);
  ProcessLevel[ProcessLevelPtr] := BlockLevel;
end;

procedure EndProcessLevel;
begind
  dec(ProcessLevelPtr);
end;

function ArrayDepth(p:integer):longint;
var tmp:word;
begind
  tmp := 0;
  while (TypeTabl^ [p].TypeDesc=ArrayType) and (tmp <= 10) do begin
inc(tmp);
p := TypeTabl^[p].p2;
end;
if tmp > 10 then FlagError(405);
ArrayDepth := tmp;
end;

function StackSize : longint;
begin
  StackSize := StackTabl^[StackTablPtr];
end;

procedure AdjustStack(Size:longint);
begin
  StackTabl^[StackTablPtr] := StackTabl^[StackTablPtr] + Size;
end;

function ExitLabel : longint;
begin
  ExitLabel := ExitTabl^[ExitTablPtr];
end;

procedure NewExitLevel(ExitLabel : longint);
begin
  inc(ExitTablPtr);
  ExitTabl^[ExitTablPtr] := ExitLabel;
  inc(StackTablPtr);
  StackTabl^[StackTablPtr] := 0;
end;

procedure EndExitLevel;
begin
  dec(ExitTablPtr);
  dec(StackTablPtr);
end;

end.
unit options;
{used to flag all options - call to check on current option settings}
OptionStatus : array['A'..'Z'] of boolean;

procedure InitOptions;
var c:char;
begin
  for c := 'A' to 'Z' do
    OptionStatus[c] := false;
  OptionStatus['T'] := true;
  OptionStatus['L'] := true;
  DefaultStack := $4000;
end;

procedure EnableOption(c:char);
begin
  OptionStatus[UpCase(c)] := true;
end;

procedure DisableOption(c:char);
begin
  OptionStatus[UpCase(c)] := false;
end;

function CheckOption(c:char):boolean;
begin
  CheckOption := OptionStatus[UpCase(c)];
end;

procedure StoreStackSize(s:longint);
begin
  DefaultStack := s;
end;

end.
unit strstore;
{used to store and retrieve string constants}

interface
uses types,errors;

procedure SaveStringConstant(s:DataString;var CurrentNum:longint);
procedure SaveRealConstant(r:double;var CurrentNum:longint);
procedure InitStringStorage;
procedureGetStringConstant(var s:DataString;StringNum:longint);
procedure GetRealConstant(var r:double;RealNum:longint);
procedure DiscardStringStorage;

implementation
const
  MaxNumChars = 4000;
  MaxNumStrings = 200;
  MaxNumReals = 200;
type
RealTableType = array[1..MaxNumReals] of double;
RealTblPtr = ^RealTableType;

CharTableType = array[1..MaxNumChars] of char;
ChrTblPtr = ^CharTableType;

StringPtrs = record
    CharPos : integer;
end;

var
RealTableIndex : longint;
RealTable : RealTblPtr;
CurrentStringCount : integer;
CurrentCharCount : integer;
CharTable : ChrTblPtr;
StringTable : array[1..MaxNumStrings] of StringPtrs;

procedure SaveStringConstant(s:DataString;var CurrentNum:longint);
var i: integer;
begin
    inc (CurrentStringCount);
    if CurrentStringCount > MaxNumStrings then FlagError(267);
    if (CurrentCharCount+length(s) + 1) > MaxNumChars then FlagError(268);
    StringTable[CurrentStringCount].CharPos := CurrentCharCount+1;
    inc (CurrentCharCount);
    CharTable^ [CurrentCharCount] := chr(length(s));
    for i := 1 to length(s) do
        CharTable^ [CurrentCharCount+i] := s[i];
    CurrentCharCount := CurrentCharCount + length(s);
    CurrentNum := CurrentStringCount;
end;

procedure SaveRealConstant(r:double;var CurrentNum:longint);
var i:integer;
begin
    inc(RealTableIndex);
    if RealTableIndex > MaxNumReals then FlagError(272);
    RealTable^ [RealTableIndex] := r;
    CurrentNum := RealTableIndex;
end;

procedure InitStringStorage;
begin
    CurrentStringCount := 0;
    CurrentCharCount := 0;
    new(CharTable);
    RealTableIndex := 1;
    new(RealTable);
    RealTable^ [1] := pi;
procedure GetStringConstant(var s:DataString; StringNum:longint);
var i, size: integer;
begin
  s := ";
  size := ord(CharTable^[StringTable[StringNum].CharPos]);
  with StringTable[StringNum] do
    for i := CharPos + 1 to CharPos + size do
      s := s + CharTable^[i];
end;

procedure GetRealConstant(var r:double; RealNum: longint);
begin
  r := RealTable^[RealNum];
end;

procedure DiscardStringStorage;
begin
  dispose(CharTable);
  dispose(RealTable);
end;

end.

unit errors;
{used to flag all errors & terminate compilation when an error occurs}

interface
uses types,errmsg;
var
  error : boolean; ErrorNum : longint;
  ErrorRow, ErrorCol, ErrorFnum : integer;
  ErrorLocated : boolean;
  ErrorMessage : string[80];
  AbortAddress : pointer;
  AbortAlias : array[0..1] of word absolute AbortAddress;

procedure FlagError(num: longint);
procedure FlagErrorPos(num: longint; row, col, fnum: integer);

implementation

procedure abort(addr: word);
inline(
  $58/          {pop ax}
  $8B/$E0/     {mov sp,ax}
  $5D/        {pop bp}
  $CA/$00/$00/; {ret 4}
)

procedure GetErrMsg;
var errfile:text; i, tmpnum: integer; found:boolean;
  tempstr:string[80];
begin
str(ErrNum,ErrorMessage);
ErrorMessage := 'Error # ' + ErrorMessage + ': ' + ErrMsgStr(ErrNum);
ErrorLocated := false;
end;

procedure FlagError(num:longint);
begin
if not error then {only flag the error if no other error has occurred}
begin
error := true; {flag the error}
ErrorNum := num; {indicate the type of error}
ErrMsg;
abort(AbortAlias[0]-6);
end;
end;

procedure FlagErrorPos(num:longint; row,col,fnum:integer);
var errfile:text; i,tmpnum:integer; found:boolean;
tempstr:string[80];
begin
if not error then {only flag the error if no other error has occurred}
begin
error := true; {flag the error}
ErrorNum := num; {indicate the type of error}
ErrorRow := row;
ErrorCol := col;
ErrorFnum := fnum;
ErrMsg;
ErrorLocated := true;
abort(AbortAlias[0]-6);
end;
end;

begin
end.
unit ErrMsg;

interface
uses dos;
function ErrMsgStr(ErrMsgNum:longint):string;

implementation

type
MsgRecType = record
  ErrorNumber : longint;
  MsgPtr : longint;
  MsgLength : byte;
end;
functionErrMsgStr(ErrorMsgNum:longint):string;
var
  MsgCount,i : longint;
  MsgRec : array[1..2000] of MsgRecType;
  ErrorFile : file;
  TmpStr : string;
  Dir : DirStr;
  Name : NameStr;
  Ext : ExtStr;
begin
  FSplit(ParamStr(0),Dir,Name,Ext);
  assign (ErrorFile,Dir+'errors' );
  {$I-}reset(ErrorFile,1);{$I+}
  if IOResult <> 0 then
    ErrMsgStr := 'Unable to Open Error Message File'
  else begin
    BlockRead(ErrorFile,MsgCount,4);
    BlockRead(ErrorFile,MsgRec[1],MsgCount*SizeOf(MsgRecType));
    i := 1;
    while (i <= MsgCount) and (MsgRec[i].ErrorNumber <> ErrorMsgNum)
      do inc(i);
    if i > MsgCount then ErrMsgStr := 'Unknown Error'
    else begin
      seek(ErrorFile,4+MsgCount*SizeOf(MsgRecType) + MsgRec[i].MsgPtr-1);
      BlockRead(ErrorFile,TmpStr[1],MsgRec[i].MsgLength);
      TmpStr[0] := chr(MsgRec[i].MsgLength);
      ErrMsgStr := TmpStr;
    end;
    close (ErrorFile);
  end;
end.

unit types;
{declares all types used}

interface
const
  MaxTypeSize = $200000;
  {any type bigger than this is an error}
  PageSize = 4096;
type
  DataString = string;

LexSymbol = (begin1,end1,program1,procedure1,function1,forward1,if1,then1,
  else1,while1,do1,for1,to1,downto1,with1,repeat1,until1,assign1,
  equal1,LessThen1,GreaterThen1,LessEql,GreaterEql,Round1,
  NotEqual1,in1,plus1,minus1,times1,divide1,case1,of1,var1,ret1,
  type1,const1,integer1,shortint1,longint1,word1,byte1,boolean1,
  char1,real1,single1,double1,string1,array1,file1,text1,set1,
record1, true1, false1, colon1, semicolon1, comma1, period1, DotDot1, 
carrot1, ampersand1, div1, mod1, LeftPar1, RightPar1, LeftBrkt1, 
RightBrkt1, and1, or1, xor1, not1, StringConst1, LongintConst1, 
RealConst1, inc1, dec1, pred1, succ1, ord1, chr1, read1, readln1, 
write1, writeln1, CharConst1, identifier1, error1, eof1, eoc1, 
fassign1, reset1, rewrite1, close1, seek1, unit1, interface1, 
implementation1, uses1, pil, absolute1, exception1, nil1, pointer1, 
setexceptionvec1, inline1, process1, resource1, parallel1, select1, 
channel1, sin1, cos1, tan1, asin1, acos1, atan1, ln1, exp1, sqrt1, abs1, 
sqr1, move1, shared1, template1, requestread1, requestwrite1, 
release1, ptr1, addr1, sizeof1, exit1, fillchar1);

LexOutput = record
  token : LexSymbol;
  i1 : longint;
end;

IdDesc = (LongintConst, CharConst, StrConst, BoolConst, EnumConst, 
  RealConst, TypeName, ProcName, FuncName, 
  VarName, ValParamName, VarParamName, RetParamName, 
  FwdProcName, FwdFuncName, UnitName, 
  DecRsrcProc, UsedRsrcProc, WithField);

IdRec = record
  IdTyp : IdDesc; {what kind of identifier is it?}
  IdNum : longint; {what is the name of the identifier?}
  level : shortint; {0=main block, etc}
  valptr : longint; {miscellaneous value/pointer storage}
end;

TypeDefDescType = (IntType, LongintType, ShortintType, ByteType, WordType, 
  RealType, DoubleType,SingleType,PtrType,FwdPtrType, 
  CharType, BoolType, StringType, ArrayType, RecordType, 
  SetType, SubrangeType, EnumType, FileType, TextType, 
  DeadFwdPtrType, ChannelType, ProcessType, 
  ResourceType, TemplateType, SharedType);

ExprDataType = record
  ExprType : TypeDescType;
  IsConst : boolean;
end;

TypeDefDataType = record
  TypeDesc : TypeDescType;
  SubType : longint;
  p1,p2 : longint; {miscellaneous pointers and/or values}
  VariableSpace : longint; {variable size for resource}
  TypeSize : longint; {size of the data type}
  AddrMask : longint; {mask used to properly allign memory addresses}
  level : shortint; {current block level}
end;
ConstResultType = record
    CType : IdDesc;
    ConstantVal : longint;
end;

FieldDescType = record
    FName,FTypePtr,FTypePcnt,AddrOffs : longint;
    row,col,fnum:integer;
end;

VarDataType = record
    VarTypePtr : integer;
    AddrOffset : longint;
    level : shortint;
    IsAbsolute : (NotAbs,AbsVar,AbsVarParam,AbsValParam);
    AbsOffset : longint;
    AbsLevel : shortint;
end;

ProcEntryType = record
    level : byte;
    FirstParam,ParameterCount : integer;
    ParamSize : longint;
    CodeLabel : longint;
    row,col,fnum : integer;
end;

FuncEntryType = record
    level : byte;
    FirstParam,ParameterCount,TypePointer : integer;
    ParamSize : longint;
    CodeLabel : longint;
    RsltOffset : longint;
    row,col,fnum : integer;
end;

ParmType = (ValPtype,VarPtype,RetPtype);

ParamEntryType = record
    TypePointer : integer;
    IsVar : ParmType;
    AddrOffs : longint;
    IdNum : integer;
    IsProcParm : boolean;
end;

OpCode = (Initialize,DecLabel,JmpTrue,JmpFalse,JmpAlways,ProgStart,ProgEnd,
    ProcStart,ProcEnd,FuncStart,FuncEnd,PrepFuncCall,FuncCall,
    ProcCall,NegateVal,PushStruct,CopyStruct,EvalCase,CastCmp,
    PushBaseVarAddr,PushBaseParamAddr,PushVal,PushVar,CopyParms,
    IncVar,DecVar,Add,Subtract,Mult,Divide,IntDiv,Modulo,ChkRange,
IntConstant, BoolConstant, RealConstant, LogAnd, LogOr, LogXor, LogNot,
Fetch, Store, AddConstant, MultConstant, ChkStrIndex,
ToCmp, DowntoCmp, ForEnd, ForInc, ForDec, EnablePar, DisablePar,
BitwiseNot, CmpEQ, CmpNE, CmpLT, CmpLE, CmpGT, CmpGE,
StrConstant, StoreStr, StoreCharInStr, PushStrValParam,
FetchStr, PushCharAsStrParam, AddStrings, AddCharStr,
AddStrChar, AddCharChar, CharToStr, ExpandFuncStr,
InitStr, CloseStr, SourceCodeEnd,
AssignFile, ResetFile, RewriteFile, CloseFile, SeekFile,
SelectStdIO, SelectIO, ReleaseIO, ChkEOF, ChkEOC, FlushIO,
PrintInt, PrintReal, PrintChar, PrintString, PrintEol, Address2Int,
ReadFile, WriteFile, CtrlInputLine, ReadVar, ExProcStart, ExProcEnd,
GetLabelAddress, StoreVector, DecWord, DecLong,
StartProcess, EndProcess, DoRound,
StartRsrcProc, EndRsrcProc, ParallelCall, CallRsrcProc, DoSin,
DoCos, DoTan, DoASin, DoACos, DoATan, DoExp, DoLn, DoSqr, DoAbs,
DoSqr, CmpStrs, CmpStrChar,
CmpCharStr, DoMove, ForPrelim, EndOfAnd, EndOfOr, DoFillChar,
InitPF, ClosePF, ChkSelect, SkipElse, ChooseSelect, PushLabelAddr,
PrepareTemplateWrite, PrepareTemplateRead, ReqRead, ReqWrite,
TempRelease, FinishTemplateCheck, TempRangeStore, FinishTemplateOp,
ChkTempIndex, ChkTempIndexRow, FixStack, Trace);

SizeType = (L,W,B);

implementation
begin
end.
Appendix B

BOS Operating System Source Code Listing

program BOS;
{
*** Make this supervisor code - ram variables start at $00400000 ***
*** EPROM space starts at $00000000 ***
{$K+} { This forces in the reset vectors - ISP & PC at reset. }
{$M $00400000,$00000000>
}
const
    TestVer = false;
    TimerBaseAddr = $00180003;
    TimerStride = 4;
    TimerIntVector = 29;

{...}

type
    PacketType = array[-12..1499] of byte;

    QueueType = record
        Head,
        Tail : longint;
        PCount : longint;
    end;

    WaitingPType = record
        WaitCount : longint;
        WaitingProcs : array[1..1022] of longint;
    end;

    MessageStrType = array[0..79] of char;

    MessageBlockType = record
        InPtr,OutPtr : byte;
        Messages : array [0..49] of MessageStrType;
    end;

    ShortDescriptorType = longint;

    LongDescriptorType = record
        Limit : word;
        DescType : word;

594
TableAddress : longint;
end;

FuncLevelDescTable = array[0..7] of ShortDescriptorType;
LevelADescTable = array[0..7] of ShortDescriptorType;
LevelBDescTable = array[0..15] of ShortDescriptorType;
LevelCDescTable = array[0..7] of ShortDescriptorType;
LevelDDescTable = array[0..1023] of ShortDescriptorType;

ProcessData = record
  A5Value,
  A6Value,
  RetAddrValue,
  PrevPtr,
  NextPtr,
  QDataPtr,
  StackStore,
  PPageBusy,
  ProcessParamAddr,
  ProcessParentID,
  ProcessParentAddr,
  ResourceNext2Exec,
  ResourceNextWait,
  ResourceEndParms,
  ResourceParentA7,
  CurrentHandle,
  NextSelectNum,
  UseMMU : longint;
  IOBuffer : string;
  MMUPages : LevelCDescTable;
end;

PProcessData = ^ProcessData;

{*** Global & Absolute Variables ***}
var
  FLDT : FuncLevelDescTable;
  SLADT,
  LADDT,
  LACDT : LevelADescTable;
  LBDDT,
  LBCDT : LevelBDescTable;
  LocalLCDDT,
  LCCDT : LevelCDescTable;
  LocalLDDDT,
  LDCDT : LevelDDescTable;

procedure DisableInterrupts(var PrevValue : word);
begin
  inline($40e7/ {move.w sr,-(a7)}
              $007c/$0700 {ori.w #$0700,st}
procedure RestoreInterrupts(PrevValue : word);
begin
inline(
    $46df;  \{move.w (a7)+,sr\}
end;

const
RamBase = $400000;
RamSize = $400000;
A24BaseAddr = $A0000000;

{ Mizar 7130 Control Registers. }
var
AddrSel : longint absolute $00180200;
MiscReg : longint absolute $00180300;
Status : longint absolute $00180400;
MonSwT : byte absolute $00180703;
IRqVect : byte absolute $00180603;

const
DefaultMiscReg = $1be84ffe;
LED1 = $01000000;
LED2 = $02000000;

procedure SCSIInterruptjexception;
begin
  ptr byte ($00180103) := $17;
  ptr byte ($00180107) := 0;
end;

procedure InitCPUBoard;
begin
  SetExceptionVec(27,@SCSIInterrupt);
end;

function MyCPUlD:longint;
begin
  MyCPUID := ((Status div $10000) AND $000f) + 1;
end;

procedure InitCPU;
begin
  { Initialize MiZAR Registers. }
  MonSwT := $7f;
  AddrSel := $01000000 + ($ff AND (MyCPUID - 1));
  MiscReg := DefaultMiscReg;
end;
procedure InterruptProcessor(CPUNum : longint);
begin
  ptr byte ($fffff000 + $100 * ($00ff AND (CPUNum - 1))) := 0;
end;

procedure SetLED(NewVal : longint; var PrevVal : longint);
begin
  PrevVal := $03000000 - (MiscReg AND $03000000);
  MiscReg := DefaultMiscReg - (NewVal AND $03000000);
end;

procedure Delay(T : longint);
var
  k, i : longint;
begin
  for i := 1 to 6 * T do
    k := (3 * i) + (i mod 7);
end;

procedure DebugMark(Count : longint);
var
  TempLED : longint;
begin
  SetLED(LED1,TempLED);
  Delay(10000);
  SetLED(0,TempLED);
  Delay(10000);
  while Count > 0 do begin
    SetLED(LED1,TempLED);
    Delay(3000);
    SetLED(0,TempLED);
    Delay(3000);
    dec(Count);
  end;
  Delay(10000);
end;

procedure HoldGlobalBus;
begin
  MiscReg := MiscReg OR $00300000;
end;

procedure ReleaseGlobalBus;
begin
  MiscReg := (MiscReg AND ($fffffiff - $00300000)) OR 00100000;
  MonSwt := $80;
end;

const
  EthernetRam = $810000; {*** Base address of Ethernet Ram ***}
XmtRam = EthernetRam;  {*** Transmit buffer located at base ***}
RcvRam = EthernetRam + $8000;  {*** Receive at mid-point ***}

EtherCPUBase = $800000;  {*** Base address of Ethernet Processor ***}

{*** 8390 LAN Controller (page0) register offset for read and write***}
CMDR = $0;  {*** command register for read & write***}
PSTART = $1;  {*** page start register for write***}
PSTOP = $2;  {*** page stop register for write***}
BNRY = $3;  {*** boundary reg for rd and wr***}
TPSR = $4;  {*** tx start page start reg for wr***}
TBCR0 = $5;  {*** tx byte count 0 reg for wr***}
TBCR1 = $6;  {*** tx byte count 1 reg for wr***}
ISR = $7;  {*** interrupt status reg for rd and wr***}
CRDA0 = $8;  {*** current remote dma address 0 for rd***}
RSAR0 = $8;  {*** remote start address reg 0 for wr***}
CRDA1 = $9;  {*** current remote dma address 1 for rd***}
RSAR1 = $9;  {*** remote start address reg 1 for wr***}
RBCR0 = $A;  {*** remote byte count reg 0 for wr***}
RBCR1 = $B;  {*** remote byte count reg 1 for wr***}
RSR = $C;  {*** rx status reg for rd***}
RCR = $C;  {*** rx configuration reg for wr***}
CNTR0 = $D;  {*** tally cnt 0 for frm alg err for rd***}
TCR = $D;  {*** tx configuration reg for wr***}
CNTR1 = $E;  {*** tally cnt 1 for crc err for rd***}
DCR = $E;  {*** data configuration reg for wr***}
CNTR2 = $F;  {*** tally cnt 2 for missed pkt for rd***}
IMR = $F;  {*** interrupt mask reg for wr***}

{*** 8390 LAN Controller (page1) register offset for read and write***}
PAR0 = $1;  {*** physical addr reg 0 for rd and wr***}
PAR1 = $2;  {*** physical addr reg 1 for rd and wr***}
PAR2 = $3;  {*** physical addr reg 2 for rd and wr***}
PAR3 = $4;  {*** physical addr reg 3 for rd and wr***}
PAR4 = $5;  {*** physical addr reg 4 for rd and wr***}
PAR5 = $6;  {*** physical addr reg 5 for rd and wr***}
CURR = $7;  {*** current page reg for rd and wr***}
MAR0 = $8;  {*** multicast addr reg 0 for rd and WR***}
MAR1 = $9;  {*** multicast addr reg 1 for rd and WR***}
MAR2 = $A;  {*** multicast addr reg 2 for rd and WR***}
MAR3 = $B;  {*** multicast addr reg 3 for rd and WR***}
MAR4 = $C;  {*** multicast addr reg 4 for rd and WR***}
MAR5 = $D;  {*** multicast addr reg 5 for rd and WR***}
MAR6 = $E;  {*** multicast addr reg 6 for rd and WR***}
MAR7 = $F;  {*** multicast addr reg 7 for rd and WR***}

{********************************************************************}
{  8003 control register operations }
{********************************************************************}

MSK_RESET = $80;  {*** reset LAN controller***}
MSK_ENASH = $40;  {*** enable PC access to shared mem***}
MSK_DECOD = $3F; {*** ??? memory decode bits, corresponding***}
{*** to SA 18-13. SA 19 assumed to be 1***}

{*******************************************************************************}

{ 8390 CMDR MASK }

{*******************************************************************************}

MSK_STP = $01; {*** software reset, take 8390 off line***}
MSK_STA = $02; {*** activate the 8390 NIC***}
MSK_TXP = $04; {*** initial txing of a frm***}
MSK_RD2 = $20; {*** abort remote DMA***}
MSK_PG0 = $00; {*** select register page O***}
MSK_PG1 = $40; {*** select register page 1***}

{*******************************************************************************}

{ 8390 ISR & IMR MASK }

{*******************************************************************************}

{ 8390 DCR MASK }

{*******************************************************************************}

MSK_PRX = $01; {*** rx with no error***}
MSK_PTX = $02; {*** tx with no error***}
MSK_RXE = $04; {*** rx with error***}
MSK_TXE = $08; {*** tx with error***}
MSK_OVW = $10; {*** overwrite warning***}
MSK_CNT = $20; {*** MSB of one of the tally counters is set***}
MSK_RDC = $40; {*** remote dma completed***}
MSK_RST = $80; {*** reset state indicator***}

{*******************************************************************************}

{ 8390 RCR MASK }

{*******************************************************************************}

MSK_WTS = $01; {*** word transfer mode selection***}
MSK_BOS = $02; {*** byte order selection***}
MSK_LAS = $04; {*** long addr selection***}
MSK_BMS = $08; {*** burst mode selection***}
MSK_ARM = $10; {*** atuoinitlize remote***}
MSK_FT00 = $00; {*** burst length selection***}
MSK_FT01 = $20; {*** burst length selection***}
MSK_FT10 = $40; {*** burst length selection***}
MSK_FT11 = $60; {*** burst length selection***}

{*******************************************************************************}

{ 8390 RCR MASK }

{*******************************************************************************}

MSK_SEP = $01; {*** save error pkts***}
MSK_AR = $02; {*** accept runt pkt***}
MSK_AB = $04; {*** accept broadcast***}
MSK_AM = $08; {*** accept multicast***}
MSK_PRO = $10; {*** promiscuous physical***}
{*** accept all pkt with physical adr***}
MSK_MON = $20; {*** monitor mode***}
{************************************************************************************************}  
{  8390 TCR MASK}  
{************************************************************************************************}  

MSK_CRC = $01; {*** inhibit CRC, do not append crc***}  
MSK_LBO1 = $06; {*** encoded loopback control***}  
MSK_ATD = $08; {*** auto tx disable***}  
MSK_OFST = $10; {*** collision offset enable***}  

{************************************************************************************************}  
{  8390 RSR MASK}  
{************************************************************************************************}  

SMK_PRX = $01; {*** rx without error***}  
SMK_CRC = $02; {*** CRC error***}  
SMK_AAE = $04; {*** frame alignment error***}  
SMK_FO = $08; {*** FIFO overrun***}  
SMK_MPA = $10; {*** missed pkt***}  
SMK_PHY = $20; {*** physical/multicase address***}  
SMK_DIS = $40; {*** receiver disable, set in monitor mode***}  
SMK_DEF = $80; {*** deferring***}  

{************************************************************************************************}  
{  8390 TSR MASK}  
{************************************************************************************************}  

SMK_PTX = $01; {*** tx without error***}  
SMK_DFR = $02; {*** non deferred tx***}  
SMK_COL = $04; {*** tx collided***}  
SMK_ABT = $08; {*** tx abort because of excessive collisions***}  
SMK_CRS = $10; {*** carrier sense lost***}  
SMK_FU = $20; {*** FIFO underrun***}  
SMK_CDH = $40; {*** collision detect heartbeat***}  
SMK_OWC = $80; {*** out of window collision***}  

var  
XmtBuff : PacketType absolute XmtRam;  
RcvBuff : array[0..$7ff] of byte absolute RcvRam;  
EtherCPU : array[0..$1e] of byte absolute EtherCPUBase;  

const  
PageSize = 4096;  
RamWords = RamSize div 4;  
RamPages = RamSize div PageSize;  
RamPagesM1 = RamPages - 1;  
UserCodeBase = $e0000000;  
UserDataBase = $c0000000;  
UserDataLevelAIndex = $07 AND (UserDataBase div $20000000);  
UserDataLevelBIndex = $0f AND (UserDataBase div $200000);  
UserDataLevelCIndex = $07 AND (UserDataBase div $400000);  
UserCodeLevelAIndex = $07 AND (UserCodeBase div $20000000);
UserCodeLevelBIndex = $0f \text{ AND (UserCodeBase div } 2000000\text{)};
UserCodeLevelCIndex = $07 \text{ AND (UserCodeBase div } 400000\text{)};
LevelBStride = 4096 \times 1024 \times 8;
UserDataLevelBIndex2 = (UserDataLevelBIndex + 1) \text{ AND } 0000000f;
ProcessIntLevel = 0;

InitialISR = $1000 + \text{RamBase};
InitialMSR = $2000 + \text{RamBase};

var
InsideEthernet,
AwaitingEtherResponse : boolean;
AckNeeded : boolean;
i : longint;
ProgAddr : longint;

ErrorMessageBuffer : array[1..2048] of char;
ErrorMessagePtr : longint;
RunTimeErrorPresent : boolean;

PageNum : array[0..\text{RamPagesM1}] of word;
FreePagePtr : longint;
CodeHighPtr : longint;
NumCodePages : longint;

ReadyQ,
ParentProcQ,
ResourceWaitQ,
OpenChanQ,
SendChanQ,
RcvChanQ,
ShareQ,
EtherQ : QueueType;

FlickAddr : array[1..25] of longint;
FlickDepth : longint;

ProgramDone : boolean;

OSMsgBlk : \text{^MessageBlockType};
ServeOSMsg : boolean;

MyWindow : longint;
MyWait : \text{^WaitingPType absolute MyWindow};
DoTrace,
DoMsg : boolean;

MainLoopLEDSave : longint;

LastVariable : byte;
procedure ResetCPU;
var
 i:longint;
 LP : ^longint;
begin
 for i := 1 to 50000 do ;
 ptr longint(addr(LP)) := addr(LP) + 4;  { LP points to return address. }
 LP := addr(ResetCPU) AND $ffff0000;
 if not TestVer then inc(LP^,8);
 { RTS at end of procedure now causes jump to beginning of kernel. }
end;

function MyProcID:longint;
var t:longint;
begin
 MyProcID := MyCPUID + (addr(t) and (-PageSize));
end;

function PDataPtr(ProcPage:longint):PProcessData;
begin
 PDataPtr := @ ptr ProcessData(
 (ProcPage AND (-PageSize)) + PageSize - SizeOf(ProcessData));
end;

function DecStr(D:longint):string;
var
 Tmp : string;
 Negate : boolean;
begin
 if D=0 then Tmp := '0'
 else begin
  Tmp := ";
  if D < 0 then begin
   Negate := true;
   D := -D;
  end else Negate := false;
  while D <> 0 do begin
   Tmp := chr((D mod 10) + ord('0')) + Tmp;
   D := D div 10;
  end;
  if Negate then Tmp := '-' + Tmp;
  DecStr := Tmp;
 end;
end;

function HexStrL(v,l:longint):string;
var t:string;c:char;x:longint;
begin
 t := ";
 while l > 0 do begin
  dec(l);
 end;
\begin{verbatim}
x := v and $0f;
v := (v and $ffffff0) div $10;
if x <= 9 then c := chr(ord('0') + x)
else c := chr(ord('A') + x - 10);
t := c + t;
end;
HexStrL := t;
end;

function LocalAddr(A:longint):boolean;
begin
  LocalAddr := (A >= RamBase) AND (A < (RamBase + RamSize));
end;

procedure FlushMessages;
begin
  if not DoMsg then Exit;
  repeat until (OSMsgBlk~.InPtr = OSMsgBlk~.OutPtr);
end;

procedure Message(S:string);
begin
  if not DoMsg then Exit;
  if (OSMsgBlk~.InPtr + 1) mod 50 <> OSMsgBlk~.OutPtr then begin
    if ord(S[0]) > 79 then S[0] := chr(79);
    OSMsgBlk~.Messages[OSMsgBlk~.InPtr] :=
      ptr MessageStrType(addr(S));
    OSMsgBlk~.InPtr := (OSMsgBlk~.InPtr + 1) mod 50;
  end;
  FlushMessages;
end;

procedure DumpProcessPage(P : longint;S : string);
var
  PDPtr: PProcessData;
i : longint;
begin
  Message(S); FlushMessages;
  P := P AND (-PageSize);
  PDPtr := PDataPtr(P);
  Message('Process Page = $' + HexStrL(P,8)
    + ' MMU Level C Address = $' + HexStrL(addr(PDPtr.MMUPages),8));
  FlushMessages;
  for i := 0 to 31 do begin
    if (i AND $0f) = 0 then S := HexStrL(i + P,8) + ': ';
    S := S + HexStrL(ptr byte(i + P),2) + ' ';
    if (i AND $0f) = $0f then begin
      Message(S);
      FlushMessages;
      S := ';
    end;
  end;
\end{verbatim}
for i := 0 to SizeOf(ProcessData) - 1 do begin
  if (i AND $0f) = 0 then S := HexStrL(i + addr(PDPtr^),8) + ' :
  S := S + HexStrLCptr byte (i + addr(PDPtr^)),2) + ':
  if (i AND $0f) = $0f then begin
    Message(S);
    FlushMessages;
    S := ";
  end;
end;
if S <> " then begin
  Message(S);
  FlushMessages;
end;
end;

function GetRamPage : longint;forward;
procedure FreeRamPage(num:longint);forward;
procedure RunTimeError(Action:longint;Message:string);forward;
procedure Flick(ReleaseCondValPtr :  longint);forward;

const
  GRamBaseAddress = $A0000000;
  GRamSize = $800000;
  GRamPageCount = GRamSize div PageSize;

var
  CPUCount: longint absolute $a0000000;
  GSendPacket: ^PacketType absolute $a0000004;
  GRecvPacket: ^PacketType absolute $a0000008;
  SharedRamStackPtr : longint absolute $a000000c;
  ExclusiveCPUDeclared : boolean absolute $a0000010;
  ExclusiveCPUNumber : longint absolute $a0000014;
  CPUWindows : array[1..16] of longint absolute $a0000018;
  CPUNumbers : array[1..16] of longint absolute $a0000058;
  SharedRamPages : array[0..GRamPageCount] of longint absolute $a0000098;

function GetGlobalPage:longint;
var
  Found : boolean;
begin
  {no interrupts allowed}
  inline($40e7/ {move.w sr,-(a7)}
    $007c/$0700); {ori.w #$0700,st}
  HoldGlobalBus;
  if SharedRamStackPtr > (GRamPageCount - 1) then Found := false
  else begin
    Found := true;
    GetGlobalPage := SharedRamPages[SharedRamStackPtr];
    inc(SharedRamStackPtr);
  end;
ReleaseGlobalBus;
{interrupts back on}
inline($46df); {move.w (a7)+,sr}
if not Found then RunTimeError(3,'Out of Global System Memory');
end;

procedure ReleaseGlobalPage(PageAddr:longint);
begin
PageAddr := (-PageSize) AND PageAddr;
if (PageAddr < GRamBaseAddress) OR (PageAddr >= GRamBaseAddress + GRamSize)
then RunTimeError(3,'Illegal Global Ram Page Release');
{no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)
$007c/$0700); {ori.w #$0700,st
HoldGlobalBus;
dec(SharedRamStackPtr);
SharedRamPages[SharedRamStackPtr] := PageAddr;
ReleaseGlobalBus;
{interrupts back on}
inline($46df); {move.w (a7)+,sr
end;

procedure IndicatePresence;
begin
MyWindow := GetGlobalPage;
MyWait^ .WaitCount := 0;
{no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)
$007c/$0700); {ori.w #$0700,st
HoldGlobalBus;
inc(CpuCount);
CPUWindows[CpuCount] := MyWindow;
CPUNumbers[CpuCount] := MyCPUId;
ReleaseGlobalBus;
{interrupts back on}
inline($46df); {move.w (a7)+,sr
end;

procedure InitGlobalRam;
var
FreeAddr : longint;
i : longint;
begin
inline($207c/> GRamBaseAddress/ {movea.l #$a0000000,a0}
$701f/ {moveq.l #31,d0}
$223c/>$0000ffff/ {L1: move.l #$fff,d1}
$4298/ {L2: clr.l (a0)+}
$51c9/$fffc/ {dbra dl,L2}
$51c8/$fff2); {dbra d0,L1
CPUCount := 0;
FreeAddr := Addr(SharedRamPages[GRamPageCount]);
FreeAddr := (FreeAddr + PageSize - 1) AND (-PageSize);
for i := (FreeAddr - GRamBaseAddress) div PageSize to GRamPageCount - 1 do
  SharedRamPages[i] := GRamBaseAddress + i * PageSize;
SharedRamStackPtr := (FreeAddr - GRamBaseAddress) div PageSize;
IndicatePresence;
ptr longint (addr (GSendPacket)) := GetGlobalPage;
ptr longint (addr (GRecvPacket)) := GetGlobalPage;
ExclusiveCPUDeclared := false;
end;

procedure FlushMMU;
begin
  inline($f000/$2400); { PFLUSHA }
end;

procedure AllocateCodePage(CAddr:longint);
begin
  while CAddr > CodeHighPtr do begin
    inc (NumCodePages);
    LDCDT[NumCodePages-1] := GetRamPage + 1;
    CodeHighPtr :=UserCodeBase + PageSize * NumCodePages - 1;
  end;
  FlushMMU;
end;

procedure TranslateGlobalAddresses(var DT : LevelDDescTable);
const
  RamEnd = RamBase + RamSize;
begine
  if A24BaseAddr = GRamBaseAddress then Exit;
  inline {
    $2054/ { MOVE.L (A4)),A0 }
    $223C/>$000003FF/ { MOVE.L #1023,D1 }
    { L: }
    $2010/ { MOVE.L (A0),D0 }$0C80/>RamBase/ { CMPI.L #RamBase,D0 }
    $6D08/ { BLT.S DOIT }$0C80/>RamEnd/ { CMPILL #RamEnd,D0 }
    $6D0C/ { BLT.S DONTDOIT }$0280/>$00FFFFFF/ { ANDI.L #$00FFFFFF,D0 }
    $0680/>A24BaseAddr/ { ADDL.I #A24BaseAddr,D0 }
    { DONTDOIT: }
    $20C0/ { MOVE.L D0,(A0)+ }$51C9/$FFDE { DBF D1,L }
  };
end;

procedure RestoreMmu(ProcessPage:longint);
var
  PDPtr : PProcessData;
LDAddr : longint;
i : longint;
begin
  for i := 0 to 15 do LBDDT[i] := 0;  { Invalid }
  LBDDT[UserDataLevelBIndex] := addr(LocalLCDDT) + 2;
  for i := 0 to 7 do LocalLCDDT[i] := 0;  { Invalid }
  LocalLCDDT[UserDataLevelCIndex] := addr(LocalLDDDT) + 2;
  PDPtr := PDataPtr(ProcessPage);
  LDAddr := PDPtr^.MmuPages[UserDataLevelCIndex] AND (-PageSize);
  LocalLDDDT := ptr LevelDDescTable (LDAddr);
  TranslateGlobalAddresses (LocalLDDDT);
  FlushMMU;
end;

procedure BWriteUP(MemAddr,val:longint);
var
  Page : longint;
begin
  Page := ((MemAddr AND (-PageSize)) - UserCodeBase) div PageSize;
  Page := LDCDT[Page] AND (-PageSize);
  ptr byte (Page + (MemAddr AND (PageSize - 1))) := Val;
end;

function LReadUP(DAddr:longint):longint;
begin
  inline($7002/ {moveq.l #2,d0}
           $4e7b/$0000/ {movec.w d0.sfc}
           $2054/ {movea.l (a4),a0}
           $0e90/$0000/ {moves.l (a0),d0}
           $2940/$0004); {move.l d0,4(a4)}
end;

function WReadUP(DAddr:longint):longint;
begin
  inline($7002/ {moveq.l #2,d0}
           $4e7b/$0000/ {movec.w d0.sfc}
           $2054/ {movea.l (a4),a0}
           $7001/ {moveq.l #0,d0}
           $0e50/$0000/ {moves.w (a0),d0}
           $2940/$0004); {move.l d0,4(a4)}
end;

const
  IndexCMask = $400000 * 7;
  IndexCDiv = $400000;
  IndexDMask = $1000 * 1023;
  IndexDDiv = $1000;

function PhysAddr(LAddr:longint):longint;
var
  PDPtr : PProcessData;
begin
  (function code...)}
procedure BlockUDRead(PAddr, LAddr, Count: longint);
var
  SmallCount: longint;
begin
  while Count > 0 do begin
    if Count <= PageSize - (LAddr and (PageSize-1))
      then SmallCount := Count
    else SmallCount := PageSize - (LAddr and (PageSize-1));
    move(ptr byte(PhysAddr(LAddr)), ptr byte(PAddr), SmallCount);
    Count := Count - SmallCount;
    PAddr := PAddr + SmallCount;
    LAddr := LAddr + SmallCount;
  end;
end;

procedure BlockUDWrite(PAddr, LAddr, Count: longint);
var
  SmallCount: longint;
begin
  while Count > 0 do begin
    if Count <= PageSize - (LAddr and (PageSize-1))
      then SmallCount := Count
    else SmallCount := PageSize - (LAddr and (PageSize-1));
    move(ptr byte(PAddr), ptr byte(PhysAddr(LAddr)), SmallCount);
    Count := Count - SmallCount;
    PAddr := PAddr + SmallCount;
    LAddr := LAddr + SmallCount;
  end;
end;

procedure BlockUDCopy(SLAddr, DLAddr, Count: longint);
var
  SCount, DCount, SmallCount: longint;
begin
  while Count > 0 do begin
    SCount := PageSize - (SLAddr and (PageSize-1));
    DCount := PageSize - (DLAddr and (PageSize-1));
    if SCount < DCount then SmallCount := SCount else SmallCount := DCount;
    if SmallCount > Count then SmallCount := Count;
  end;
end;
move(ptr byte(PhysAddr(SLAddr)),ptr byte(PhysAddr(DLAddr)),SmallCount);
Count := Count - SmallCount;
SLAddr := SLAddr + SmallCount;
DLAddr := DLAddr + SmallCount;
end;
end;

function LReadUD(DAddr:longint):longint;
var
  Tmp : longint;
begin
  if (DAddr and (PageSize-1)) + 3 < PageSize then
    LReadUD := ptr longint(PhysAddr(DAddr))
  else begin
    BlockUDRead(addr(Tmp),DAddr,sizeof(longint));
    LReadUD := Tmp
  end;
end;

function WReadUD(DAddr:longint):longint;
var
  Tmp : word;
begin
  if (DAddr and (PageSize-1)) + 1 < PageSize then
    WReadUD := ptr word(PhysAddr(DAddr))
  else begin
    BlockUDRead(addr(Tmp),DAddr,sizeof(word));
    LReadUD := Tmp
  end;
end;

function BReadUD(DAddr:longint):longint;
begin
  BReadUD := ptr byte(PhysAddr(DAddr));
end;

procedure LWriteUD (DAddr,Data:longint);
begin
  if (DAddr and (PageSize-1)) + 3 < PageSize then
    ptr longint(PhysAddr(DAddr)) := Data
  else
    BlockUDWrite(addr(Data),DAddr,sizeof(longint));
end;

procedure WWriteUD(DAddr,Data :longint);
var Tmp : word;
begin
  if (DAddr and (PageSize-1)) + 1 < PageSize then
    ptr word(PhysAddr(DAddr)) := Data
  else begin
    Tmp := Data;
procedure BWriteUD(DAddr, Data: longint);
begin
  ptr byte (PhysAddr(DAddr)) := Data
end;

function RemotePhysAddr(PPage, LAddr: longint): longint;
var
  PDPtr : PProcessData;
  Page : longint;
  IndexC, IndexD : longint;
begin
  PDPtr := PDataPtr(PPage);
  IndexC := (IndexCMask AND LAddr) div IndexCDiv;
  IndexD := (IndexDMask AND LAddr) div IndexDDiv;
  Page := (-PageSize) AND PDPtr^..MmuPages[IndexC];
  Page := ptr LevelDDescTable(Page) [IndexD] AND (-PageSize);
  RemotePhysAddr := Page + (LAddr AND (PageSize - 1));
end;

procedure Proc2ProcCopy(P1Page, P1Addr, P2Page, P2Addr, Count: longint);
var
  C1, C2, Ci : longint;
begin
  while Count <> 0 do begin
    C1 := PageSize - (P1Addr and (PageSize - 1));
    C2 := PageSize - (P2Addr and (PageSize - 1));
    if C1 > C2 then Ci := C2 else Ci := C1;
    if Ci > Count then Ci := Count;
    move(ptr byte(RemotePhysAddr(P1Page, P1Addr)),
         ptr byte(RemotePhysAddr(P2Page, P2Addr)),
         Ci);
    Count := Count - Ci;
    P1Addr := P1Addr + Ci;
    P2Addr := P2Addr + Ci;
  end;
end;

procedure ReleasePages(PPage: longint);
var
  TempVar,
  i, j,
  MmuPage : longint;
  PDPtr : PProcessData;
begin
  PDPtr := PDataPtr(PPage);
  for j := 0 to 7 do
    if PDPtr^.MmuPages[j] <> 0 then begin
MmuPage := (-PageSize) AND PDPtr^.MmuPages[j];
for i := 0 to 1023 do
  if ptr LevelDDescTable(MmuPage)[i] <> 0 then
    FreeRamPage(ptr LevelDDescTable(MmuPage)[i]);
    FreeRamPage(MmuPage);
end;

{ What we would like to do here is the following:
  FreeRamPage(PPage);
  Flick(0);
}
* This, however, could cause problems, since another process could
  interrupt and request the page we are releasing before we can
  Flick. This could overwrite our current stack, causing problems.
* Therefore, we will Flick first, using the stack information stored
  for normal Flicks. Then, however, we will call the routine to
  Free up the page. Finally, we will return into the waiting process,
  as in a standard Flick.
* Note that the process page was passed as a parameter. This can be
  legally used prior to the call to FreeRamPage even if the stack pointer
  is changed, since parameters are referenced to a4.}

{ No interrupts allowed. }
inline($40e7/ {move.w sr,-(a7)}
$007c/$0700); {ori.w #$0700,sta)}
{ The current process state is now saved. Now, get the return
  address of the interrupted process so that we can return to it}
TempVar := FlickAddr[FlickDepth];
dec(FlickDepth);
{ interrupts back on}
inline($46df); {move.w (a7)+,sr}
{ now, restore a7}
inline($206d/$0004/ {movea.l 4(a5),a0}
$2e48); {movea.l a0,a7}
{ Free up the ram page - all ram used by the process is now released.}
FreeRamPage(PPage);
{ finally, restore the other registers and Flick.}
inline($2a5f/ {movea.l (a7)+,a5}
$4e75); {rts}
end;

procedure InitMMU;
var
  CpuRoot : LongDescriptorType;
  TC,
  TTO,
  TT1 : longint;
i : longint;
begin
  { Set the TC Register to disable MMU. }
  TC := $00000000 + $1000000 + $000000 + $3000 + $400 + $30 + $0a;
  inline($f02d/$4000/$000c); {PMOVE.L $0c(A5),TC}
end;
{ Build all the tables in ram. }
for i := 0 to 7 do
  FLDT[i] := 0; { Invalid. }
  FLDT[1] := addr(LADDT) + 2;
  FLDT[2] := addr(LACDT) + 2;
  FLDT[5] := addr(SLADT) + 2;
  FLDT[6] := addr(SLADT) + 2;
for i := 0 to 7 do SLADT[i] := $20000000 * i + 1;
SLADT[7 AND (GRamBaseAddress div $20000000)] := A24BaseAddr + 1;

for i := 0 to 7 do LADDT[i] := 0; { Invalid. }
LADDT[UserDataLevelAIndex] := addr(LBDDT) + 2;
for i := 0 to 15 do LBDDT[i] := 0; { Invalid }
for i := 0 to 7 do LACDT[i] := 0; { Invalid. }
LACDT[UserCodeLevelAIndex] := addr(LBCDT) + 2;
for i := 0 to 15 do LBCDT[i] := 0; { Invalid }
LBCDT[UserCodeLevelBIndex] := addr(LCCDT) + 2;
for i := 0 to 7 do LCCDT[i] := 0; { Invalid }
LCCDT[UserCodeLevelCIndex] := addr(LDCDT) + 2;
for i := 0 to 1023 do LDCDT[i] := 0; { Invalid }

{ Initialize the CRP Register. }
CpuRoot.Limit := $8000; { No Limit }
CpuRoot.DescType := $0002; { Points to 4-byte Format Descriptors. }
CpuRoot.TableAddress := addr(FLDT);
inline($f02d/$4c00/$0004); {PMOVE.Q 4(A5),CRP}

{ Initialize the TT0 Register. }
{ TT0 := $ffff8173; }
TT0 := $00000000;
inline($f02d/$0800/$0010); {PMOVE.L $10(A5),TT0}

{ Initialize the TT1 Register. }
TT1 := 0;
inline($f02d/$0c00/$0014); {PMOVE.L $14(A5),TT0}

{ Initialize the TC Register. }
TC := $80000000 + $1000000 + $c00000 + $3000 + $400 + $30 + $0a;
inline($f02d/$4000/$0004); {PMOVE.L $0c(A5),TC}
end;

procedure DisableMMU;
var
  TC : longint;
begin
  { Set the TC Register to disable MMU. }
  TC := $00000000 + $1000000 + $c00000 + $3000 + $400 + $30 + $0a;
  inline($f02d/$4000/$0004); {PMOVE.L $04(A5),TC}
end;
const
RegACR = TimerBaseAddr + TimerStride * 4;
RegIMR = TimerBaseAddr + TimerStride * 5;
RegCTUR = TimerBaseAddr + TimerStride * 6;
RegCTLR = TimerBaseAddr + TimerStride * 7;
RegIVR = TimerBaseAddr + TimerStride * 12;
RegStartCounter = TimerBaseAddr + TimerStride * 14;
RegStopCounter = TimerBaseAddr + TimerStride * 15;

CounterPreloadVal = 11520; { = 0.05 sec }
CounterPreloadValHi = CounterPreloadVal div 16;
CounterPreloadValLo = CounterPreloadVal mod 16;

ShortPreloadVal = 400;
ShortPreloadValHi = ShortPreloadVal div 16;
ShortPreloadValLo = ShortPreloadVal mod 16;

procedure Relinquish; forward;

procedure DelayNOP;
var
  i : longint;
begin
  for i := 1 to 10 do ;
end;

procedure TimerStart;
begin
  { Stop the Timer }
  if ptr byte (RegStopCounter) <> 0 then ;
  DelayNOP;
  { Load the Counter }
  ptr byte (RegCTUR) := CounterPreloadValHi;
  DelayNOP;
  ptr byte (RegCTLR) := CounterPreloadValLo;
  DelayNOP;
  { Start the Timer }
  if ptr byte (RegStartCounter) <> 0 then ;
end;

procedure TimerStop;
begin
  { Stop the Timer }
  if ptr byte (RegStopCounter) <> 0 then ;
end;

{$Z+}
procedure TimerInterrupt(FormatVector:word;ProgCntr:longint;StatusReg:word;
a7,a6,a5,a4,a3,a2,a1,a0,
d7,d6,d5,d4,d3,d2,d1,d0 : longint);exception;
const
FPRegSize = 8 * 12;
FPControlSize = 3 * 4;
FPStateSize = 216; { Maximum size required for FSAVE instruction. }

var
SR : word;
TempAddrVar : longint;
FPRegSave : array[1..FPRegSize] of byte;
FPControlSave : array[1..FPControlSize] of byte;
FPStateSave : array[1..FPStateSize] of byte;

begin
if (StatusReg and $2000) <> 0 then begin
    {*** Some other OS function is interrupted. Restart the counter
    for a brief time interval and return. ***}
    { Stop the Timer }
    if ptr byte (RegStopCounter) <> 0 then ;
    DelayNOP;
    { Load the Counter }
    ptr byte (RegCTUR) := ShortPreloadValHi;
    DelayNOP;
    ptr byte (RegCTRL) := ShortPreloadValLo;
    DelayNOP;
    { Start the Timer }
    if ptr byte (RegStartCounter) <> 0 then ;
end else begin
    {* Set the system status register to be identical to that of the
    interrupted process, except that the Supervisor/User state bit
    is set. The Interrupt Priority Mask, however, will be the same
    as that of the interrupted process. In this way, future timer
    interrupts will be recognized. *}
    SR := StatusReg OR $2000;
    inline($46ed/$0004); { move.w 4(a5),SR }

    {* Save the context of the FPCPU. *}
    TempAddrVar := addr(FPStateSave);
    inline($206d/$0006/ { movea.l 6(a5), a0 }
             $f310); { fsave (a0) }
    TempAddrVar := addr(FPControlSave);
    inline($206d/$0006/ { movea.l 6(a5), a0 }
             $f210/$bc00); { fmovem FPCR/FPSR/FPIAR, (a0) }
    TempAddrVar := addr(FPRegSave);
    inline($206d/$0006/ { movea.l 6(a5), a0 }
             $f210/$f0ff); { fmovem FP0-7, (a0) }
    Relinquish;

    {* Restore the context of the FPCPU. *}
    TempAddrVar := addr(FPRegSave);
    inline($206d/$0006/ { movea.l 6(a5), a0 }
             $f210/$d0ff); { fmovem (a0), FP0-7 }
    TempAddrVar := addr(FPControlSave);
    inline($206d/$0006/ { movea.l 6(a5), a0 }
             $f210/$f0ff); { fmovem FP0-7, (a0) }

procedure InitializeTimer;
begin
{ Stop the Timer }
if ptr byte (RegStopCounter) <> 0 then
DelayNOP;
{ Specify Counter mode, CLK = Crystal div 16 }
ptr byte (RegACR) := $30;
DelayNOP;
{ Initialize the IVR }
ptr byte (ReglVR) := TimerIntVector;
DelayNOP;
{ Set the vector table up }
SetExceptionVec(TimerIntVector, @TimerInterrupt);
DelayNOP;
{ Unmask the appropriate bit in the IMR }
ptr byte (RegIMR) := $08;
end;

function GetFirstProcess(var Q:QueueType): longint;
begin
if Q.Head = addr(Q.Tail) then GetFirstProcess := -1
else GetFirstProcess := Q.Head and (-PageSize);
end;

function GetNextProcess(var Q:QueueType; StartPoint: longint): longint;
var
  PDPtr : PProcessData;
begin
  PDPtr := PDataPtr(StartPoint);
  if PDPtr^.PrevPtr = addr(Q.Tail) then GetNextProcess := -1
  else GetNextProcess := PDPtr^.PrevPtr and (-PageSize);
end;

procedure InitQ(var Q : QueueType);
begin
  Q.Head := addr(Q.Tail);
  Q.Tail := addr(Q.Head);
  Q.PCount := 0;
end;

procedure PutOnQ(var Q : QueueType; ProcessPage : longint);
var
  MyNextPtr,
MyPrevPtr : longint;
PDPtr : PProcessData;
begin
  PDPtr := PDataPtr(ProcessPage);
  {* get addresses of links within the process}
  MyNextPtr := addr(PDPtr^.NextPtr);
  MyPrevPtr := addr(PDPtr^.PrevPtr);
  {* no interrupts allowed}
  inline($40e7/ {move.w sr,-(a7)}
         $007c/$0700); {ori.w #$0700, st}
  ptr longint (MyNextPtr) := Q.Tail;
  ptr longint (Q.Tail) := MyNextPtr;
  Q.Tail := MyPrevPtr;
  ptr longint (MyPrevPtr) := addr(Q.Tail);
  inc(Q.PCount);
  {* interrupts back on}
  inline($46df); {move.w (a7)+,sr}
end;

procedure JoinQFlick(var Q : QueueType; ReleaseCondValPtr : longint);
var
  TempVar : longint;
  PDPtr : PProcessData;
  MyNextPtr,
  MyPrevPtr,
  ProcessA5, { These must be the last two parameters. }
  ProcessA6 : longint; { The caller's return address follows them. }
begin
  PDPtr := PDataPtr(addr(PDPtr));
  {* store data, stack ptrs into process data area}
  PDPtr^.QDataPtr := ReleaseCondValPtr;
  PDPtr^.StackStore := addr(ProcessA5);
  {* save values of a5 & a6 into appropriate variables, which will be
  on top of the stack when the process is restarted}
  ProcessA5 := ptr longint (addr(TempVar) - 4);
  inline($2b4e/$0004); {move.l a6,4(a5)}
  ProcessA6 := TempVar;
  {* get addresses of links within this process}
  MyNextPtr := addr(PDPtr^.NextPtr);
  MyPrevPtr := addr(PDPtr^.PrevPtr);
  {* no interrupts allowed}
  inline($40e7/
         {move.w sr,-(a7)}
        $007c/$0700); {ori.w #$0700, st}
  ptr longint (MyNextPtr) := Q.Tail;
  ptr longint (Q.Tail) := MyNextPtr;
  Q.Tail := MyPrevPtr;
  ptr longint (MyPrevPtr) := addr(Q.Tail);
  inc(Q.PCount);
  {* The current process is now saved on the queue. Now, get the return
  address of the interrupted process so that we can return to it}
  TempVar := FlickAddr[FlickDepth];
dec(FlickDepth);
inline($204f/$0004);
{movea.l a7,a0}
{movea.l 4(a5),a7}
/* interrupts back on*/
inline($46d8);
{move.w (a0)+,sr}
/* now, restore saved registers and return to original process*/
inline($2a5f/$4e75);
{movea.l (a7)+,a5}
{movea.l (a7)+,a6}
{rts}
end;

procedure Flick;(ReleaseCondValPtr : longint) - Forward declared
/* This procedure should only be called by a process that is already on 
a queue. It will then remain on that queue.*/
var
TempVar : longint;
PDPtr : PProcessData;
ProcessAS,
ProcessA6 : longint;
begin
if (FlickDepth=0) then begin
if not RunTimeErrorPresent then RunTimeError(3,'Illegal Process Flick');
repeat until not RunTimeErrorPresent;
{ ResetCPU; }
end;
/* store data, stack ptrs into process data area*/
PDPtr := PDataPtr(addr(PDPtr));
PDPtr^ .QDataPtr := ReleaseCondValPtr;
PDPtr^ .StackStore := addr(ProcessAS);
/* save values of a5 & a6 into appropriate variables, which will be 
on top of the stack when the process is restarted*/
ProcessAS := ptr longint (addr(TempVar) - 4);
inline($2b4e/$0004); {moved a6,4(a5)}
ProcessA6 := TempVar;
/* no interrupts allowed*/
inline($40e7/$0004);
{move.w sr,-(a7)}
{ori.w #$0700,st}
/* The current process state is now saved. Now, get the return 
address of the interrupted process so that we can return to it*/
TempVar := FlickAddr[FlickDepth];
dec(FlickDepth);
inline($204f/$0004);
{movea.l a7,a0}
{movea.l 4(a5),a7}
/* interrupts back on*/
inline($46d8);
{move.w (a0)+,sr}
/* mark the old page as not busy - used for global process pages. */
PDPtr^ .PPageBusy := 0;
/* now, restore saved registers and return to original process*/
inline($2a5f/$4e75);
{movea.l (a7)+,a5}
{movea.l (a7)+,a6}
{rts}
procedure LeaveQ(var Q : QueueType);
{ This procedure should only be called by a process that is already on
a queue. That process should then join another queue and flick.}
var MyNextPtr,
    MyPrevPtr : longint;
PDPtr : PProcessData;
begin
{* get addresses of links within this process}
PDPtr := PDataPtr(addr(PDPtr));
MyNextPtr := addr(PDPtr ^.NextPtr);
MyPrevPtr := addr(PDPtr ^.PrevPtr);
{* no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)}
$007c/$0700); {ori.w #$0700,st}
ptr longint (ptr longint (MyPrevPtr)) := ptr longint (MyNextPtr);
ptr longint (ptr longint (MyNextPtr)) := ptr longint (MyPrevPtr);
dec(Q.PCount);
{* interrupts back on}
inline($46df); {move.w (a7)+,sr}
end;

procedure TakeOffQ(var Q : QueueType;PPage:longint);
{* This procedure is used to take a process off a queue. The calling
process should immediately place the process onto another queue.}
var MyNextPtr,
    MyPrevPtr : longint;
PDPtr : PProcessData;
begin
{* get addresses of links within this process}
PDPtr := PDataPtr(PPage);
MyNextPtr := addr(PDPtr ^.NextPtr);
MyPrevPtr := addr(PDPtr ^.PrevPtr);
{* no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)}
$007c/$0700); {ori.w #$0700,st}
ptr longint (ptr longint (MyPrevPtr)) := ptr longint (MyNextPtr);
ptr longint (ptr longint (MyNextPtr)) := ptr longint (MyPrevPtr);
dec(Q.PCount);
{* interrupts back on}
inline($46df); {move.w (a7)+,sr}
end;

function QReady(var Q:QueueType) : boolean;
begin
QReady := Q.PCount <> 0;
end;

var
SSS : string;
procedure ExecQ(var Q:QueueType);
{ This procedure should only be executed on a queue that is known to contain
  a process. The first process on the queue will be executed, using its
  own local stack space, etc.}
var
  TempVar,
  iii : longint;
  PDPtr : PProcessData;
  LDPtr : LevelDDescTable;
  CallersA5,
  CallersA6 : longint;
begins
  { Store the values of A5 and A6 used by the caller. *}
  CallersA5 := ptr longint(addr(TempVar)-4);
  inline($2b4e/$0004); {move.l a6,4(a5)}
  CallersA6 := TempVar;
  { no interrupts allowed}
  inline($40e7/$007c); {ori.w #$0700, st}
  { Save the address of the interrupted process so that we can return to it}
  inc(FlickDepth);
  FlickAddr[FlickDepth] := addr(CallersA5);
  { interrupts back on}
  inline($46df); {move.w (a7)+,sr}
  { See if the process to be executed is being run by the main program.
    If it is, clear & reinitialize the MMU AND start the timeout counter}
  PDPtr := PDataPtr(Q.Head);
  if (FlickDepth=1) AND (PDPtr^.UseMMU = 1) AND (addr(Q) = addr(ReadyQ))
  then begin
    RestoreMmu(Q.Head and (-PageSize));
    TimerStart;
  end;
  { Get the stack pointer value for the process to execute.}
  TempVar := PDPtr^.StackStore;
  { now, restore saved registers and return to original process}
  inline($2e6d/$0004/ {movea.1 4(a5),a7}
          $2a5f/ {movea.l (a7)+,a5}
          $2c5f/ {movea.1 (a7)+,a6}
          $4e75); {rts}
end;

procedure SwapProcessOnQ(CurrPage, NewPage:longint);
var
  NewNextPtr,
  NewPrevPtr,
  CurrNextPtr,
  CurrPrevPtr : longint;
  NewDPtr,
CurrDPtr : PProcessData;

begin
{* Calculate the pointer addresses.}
NewDPtr := PDataPtr(NewPPage);
CurrDPtr := PDataPtr(CurrPPage);
NewNextPtr := addr(NewDPtr^.NextPtr);
CurrNextPtr := addr(CurrDPtr^.NextPtr);
NewPrevPtr := addr(NewDPtr^.PrevPtr);
CurrPrevPtr := addr(CurrDPtr^.PrevPtr);

{* no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)}
    $007c/$0700); {ori.w #$0700, st}
ptr longint(NewNextPtr) := ptr longint(CurrNextPtr);
ptr longint(NewPrevPtr) := ptr longint(CurrPrevPtr);
ptr longint(ptr longint(CurrNextPtr)) := NewNextPtr;
ptr longint(ptr longint(CurrPrevPtr)) := NewPrevPtr;

{* interrupts back on}
inline($46df); {move.w (a7)+, sr}
end;

procedure Relinquish;
begin
LeaveQ(ReadyQ);
JoinQFlick(ReadyQ, 0);
end;

function CreateProcessSpace(ProgAddr, PType: longint): longint;
{* Possible values of PType : 
  0 = program, 1 = process}
var
    UserStackPtr,
    ParamSize,
    FirstParamPage,
    NumPages,
    i,
    CodePtr,
    ProcessPage : longint;
PDPtr : PProcessData;
LDPtr : ^LevelDDescTable;
MStr : string;

procedure CodeWord(w: word);
begin
    ptr word(CodePtr) := w;
    CodePtr := CodePtr + SizeOf(word);
end;

procedure CodeLong(l: longint);
begin
    ptr longint(CodePtr) := l;
    CodePtr := CodePtr + SizeOf(longint);
begin
  if (PType=0) and ((ProgAddr and $f0000000) <> UserCodeBase) then begin
    /* Get page to use for stack. */
    ProcessPage := GetRamPage;
    PDPtr := PDataPtr(ProcessPage);
    /* Mark the initial stack pointer to be used. */
    PDPtr^.StackStore := addr(PDPtr^.A5Value);
    /* Clear the IO Buffer for the process. */
    PDPtr^.IOBuffer[0] := #0;
    PDPtr^.NextSelectNum := 0;
    /* Indicate that the MMU won't be used. */
    PDPtr^.UseMMU := 0;
    /* Store the initial values for a5 & a6 */
    PDPtr^.A5Value := 0;
    PDPtr^.A6Value := 0;
    /* Store the initial PC onto the stack for a rts */
    PDPtr^.RetAddrValue := ProgAddr;
    CreateProcessSpace := ProcessPage;
  end else begin
    if PType=0 then begin /* main program */
      /* Get the required pages. */
      ProcessPage := GetRamPage;
      PDPtr := PDataPtr(ProcessPage);
      /* Store the MMUPage pointer into the process page. */
      for i := 0 to 7 do PDPtr^.MMUPages[i] := 0;
      ptr longint (addr(LDPtr)) := GetRamPage;
      PDPtr^.MMUPages[ UserDataLevelCIndex ] := addr(LDPtr^) + 2;
      /* Clear the MMU Page */
      for i := 0 to 1023 do LDPtr^[i] := 0;
      /* Determine the number of bytes for stack & variables */
      UserStackPtr := LReadUP(LReadUP(ProgAddr+2) + UserCodeBase - 2) +
      UserDataBase;
      /* Determine the number of pages needed for data */
      NumPages := (UserStackPtr - UserDataBase + PageSize - 1) div PageSize;
      /* Get Pages: */
      for i := 0 to NumPages - 1 do LDPtr^[i] := GetRamPage + 1;
      /* Clear the IO Buffer for the process */
      PDPtr^.IOBuffer[0] := #0;
      PDPtr^.NextSelectNum := 0;
      /* Indicate that the MMU will be used. */
      PDPtr^.UseMMU := 1;
      /* Mark the initial stack pointer to be used. */
      PDPtr^.StackStore := addr(PDPtr^.A5Value);
      /* Store initial values of a5 & a6 onto "stack" */
      PDPtr^.A5Value := 0;
      PDPtr^.A6Value := 0;
      /* Store the initial PC onto the stack for a rts */
      PDPtr^.RetAddrValue := ProcessPage;
    end;
end;
{* Move code into the beginning of the process page.}
CodePtr := ProcessPage;
CodeWord($207c);CodeLong(UserStackPtr); {move.l #UserStackPtr,a0}
CodeWord($4e60); {move.l a0,usp}
CodeLong($3f3c0000); {move.w #0,-(a7)} {frame/vector}
CodeWord($2f3c);CodeLong(ProgAddr); {move.1 #ProgAddr,-(a7) ;ret addr}
CodeWord($3f3c);
if DoTrace then
  CodeWord($9000 + $100 * ProcessIntLevel)
  {move.w #$9000,-(a7)} {status reg for trace}
else
  CodeWord($1000 + $100 * ProcessIntLevel);
  {move.w #$1000,-Ca7)} {status reg no trace}
CodeWord($4e73); {rte}
CreateProcessSpace := ProcessPage;
end;

if PType=l then begin {Process}
  {* Get the required pages.*}
  ProcessPage := GetGlobalPage;
PDPtr := PDataPtr(ProcessPage);
  {* Store the MMUPage pointer into the process page.*}
  for i := 0 to 7 do PDPtr^ .MMUPages[i] := 0;
  ptr longint (addr(LDPtr)) := GetGlobalPage;
PDPtr^ .MMUPages[UserDataLevelCIndex] := addr(LDPtr^) + 2;
  {* Clear the MMU Page*}
  for i := 0 to 1023 do LDPtr^ [i] := 0;

  {* Get initial value of user stack pointer*}
  UserStackPtr := LReadUP(ProgAddr+6) + UserDataBase;
  {* Adjust user stack pointer to point to next page.*}
  UserStackPtr := (UserStackPtr + PageSize - 1) AND (-PageSize);
  {* Determine the size of the process parameters.*}
  ParamSize := 0;
  for i := 0 to WReadUP(ProgAddr+10) - 1 do begin
    ParamSize := ParamSize + LReadUP(ProgAddr + 14 + 6 * i);
    ParamSize := $fffffffe and (ParamSize + 1);
  end;
  {* Determine the number of pages needed for the parameters*}
  NumPages := (ParamSize + PageSize - 1) div PageSize;
  {* Determine the first page to be used for the parameters.*}
  FirstParamPage := (UserStackPtr - UserDataBase) div PageSize;
  {* Get Pages:*}
  for i := FirstParamPage to FirstParamPage + NumPages - 1 do
    LDPtr^ [i] := GetGlobalPage + 1;
  {* Clear the IO Buffer for the process.*}
  PDPtr^ .IOBuffer[0] := #0;
  PDPtr^ .NextSelectNum := 0;
  {* Indicate that the MMU will be used.*}
  PDPtr^ .UseMMU := 1;
  {* Store the address for the process parameters into the appropriate
position in the process page. }
PDPtr^.ProcessParamAddr := UserStackPtr;
{ Mark the initial stack pointer to be used. }
PDPtr^.StackStore := addr(PDPtr^.A5Value);
{ Store initial values of a5 & a6 onto "stack" }
PDPtr^.A5Value := 0;
PDPtr^.A6Value := 0;
{ Store a ret addr = ProcessPage onto stack. We'll put code there }
PDPtr^.RetAddrValue := ProcessPage;
{ Move code into the beginning of the process page. }
CodePtr := ProcessPage;
CodeWord($207c); {movea.1 #UserStackPtr,a0}
CodeLong(UserStackPtr);
CodeWord($4e60); {move.l a0,usp}
CodeWord($2848); {movea.1 a0,a4} {params addr}
CodeLong($3f3c0000); {move.w #0,-(a7)} {frame/vector}
CodeWord($2f3c); {move.l #ProgAddr,-(a7);ret addr}
CodeLong(ProgAddr);
CodeWord($3f3c);
if DoTrace then
  CodeWord($9000 + $100 * ProcessIntLevel)
    {move.w #$9000,-(a7)} {status reg for trace}
else
  CodeWord($1000 + $100 * ProcessIntLevel);
    {move.w #$1000,-(a7)} {status reg}
CodeWord($4e73); {rte}
end;
CreateProcessSpace := ProcessPage;
end;
end;

procedure InitLocalProcess(PPage : longint);
var
  ParamAddress,
  i : longint;
  PDPtr : PProcessData;
  LDPtr : ^LevelDDescTable;
begin
  PDPtr := PDataPtr(PPage);
  { Get the address of the parameters. }
  ParamAddress := PDPtr^.ProcessParamAddr;

  { Request local ram pages for those pages below the parameter pages. }
  ptr longint (addr(LDPtr)) := PDPtr^.MMUPages[UserDataLevelCIndex]
  AND (-PageSize);
  for i := 0 to (ParamAddress - UserDataBase) div PageSize - 1 do
    LDPtr^[i] := GetRamPage + 1;
end;

procedure RemoveLocalPages(PPage : longint);
var
ParamAddress,
i : longint;
PDPtr : PProcessData;
LDPtr : ^LevelDDescTable;
begin
  PDPtr := PDataPtr(PPage);
  {* Get the address of the parameters. *}
  ParamAddress := PDPtr^.ProcessParamAddr;

  {* Release local ram pages for those pages below the parameter pages. *}
  ptr longint (addr(LDPtr)) := PDPtr^.MMUPages[UserDataLevelCIndex]
  AND (-PageSize);
  for i := 0 to (ParamAddress - UserDataBase) div PageSize - 1 do begin
    FreeRamPage(LDPtr^[i]);
    LDPtr^[i] := 0;
  end;
end;

procedure Longint2Packet(Num:longint;index:longint);
var
  i:longint;
  alias : array[0..3] of byte absolute Num;
begin
  for i := 0 to 3 do GSendPacket^[i+index] := alias[3-i];
end;

function Packet2Longint(Index:longint):longint;
var
  i : longint;
  Alias : array [0..3] of byte;
begin
  for i := 0 to 3 do Alias[3-i] := GRecvPacket^[i+index];
  Packet2Longint := ptr longint (addr(Alias));
end;

var
  RcvPage : longint;

function RcvByte(P : longint):byte;
begin
  RcvByte := GRecvPacket^[P];
end;

procedure DecodeLine;
var BCount,BAddr,i : longint;
  ProgramPage : longint;

procedure GetBCount;
begin
  BCount := 0;
  for i := 3 to 4 do
begin
  BCount := BCount * $10;
  if RcvByte(i) > ord('9') then
    BCount := BCount + RcvByte(i) - ord('9') + 10
  else BCount := BCount + RcvByte(i) - ord('0');
end;
BCount := 2 * BCount;
end;

procedure GetBaddr;
var max: longint;
begin
  BAddr := 0;
  max := 6 + 2 * (RcvByte(2) - ord('0'));
  if RcvByte(2) = ord('9') then max := BCount + 2;
  for i := 5 to max do
    begin
      BAddr := $10 * BAddr;
      if RcvByte(i) > ord('9') then
        BAddr := BAddr + RcvByte(i) - ord('9') + 10
      else BAddr := BAddr + RcvByte(i) - ord('0');
    end;
  if RcvByte(2) = ord('9') then ProgAddr := BAddr;
  if RcvByte(2) <> ord('9') then
    begin
      i := 7 + 2 * (RcvByte(2) - ord('0'));
      BCount := BCount - 4 - 2 * (RcvByte(2) - ord('0'))
    end;
end;

procedure StoreBytes;
var Data: byte;
begin
  while BCount > 0 do
    begin
      Data := 0;
      if RcvByte(i) > ord('9') then
        Data := $10 * (RcvByte(i) - ord('9')) + 10
      else Data := $10 * (RcvByte(i) - ord('0'));
      inc(i); dec(BCount);
      if RcvByte(i) > ord('9') then
        Data := Data + RcvByte(i) - ord('9') + 10
      else Data := Data + RcvByte(i) - ord('0');
      inc(i); dec(BCount);
      if (BAddr and $f0000000) = UserCodeBase then begin
        if BAddr > CodeHighPtr then AllocateCodePage(BAddr);
        BWriteUP(BAddr, data);
      end else ptr byte(BAddr) := data;
      inc(BAddr);
    end;
begin
if RcvByte(1) = ord('S') then
  case RcvByte(2)-ord('0') of
    0,5 : ; {no action here, but no error either}
    1,2,3 : begin
      GetBCount;
      GetBAddr;
      StoreBytes;
      end;
    9 :  begin
      GetBCount;
      GetBAddr;
      ProgramPage := CreateProcessSpace(ProgAddr,0); {0=program}
      PutOnQ(ReadyQ,ProgramPage);
      end;
      else RunTimeError(3,'Illegal S-Record Line : S' + chr(RcvByte(2)));
      end;
end;

procedure DecodeLine2;
var BCount,BAddr,data,i : longint;
    ProgramPage : longint;
begin
  BAddr := 0;
  for i := 4 downto 1 do BAddr := $100 * BAddr + RcvByte(i);
  BCount := 0;
  for i := 8 downto 5 do BCount := $100 * BCount + RcvByte(i);
  if BCount <> 0 then
    for i := 0 to BCount-1 do begin
      data := RcvByte(i+9);
      if (BAddr and $f0000000) = UserCodeBase then begin
        if BAddr > CodeHighPtr then AllocateCodePage(BAddr);
        BWriteUP(BAddr,data);
      end else ptr byte (BAddr) := data;
      inc(BAddr);
    end
  else begin
    ProgAddr := BAddr;
    ProgramPage := CreateProcessSpace(BAddr,0); {0=program}
    PutOnQ(ReadyQ,ProgramPage);
  end;
end;

procedure PostMessageToEthernet;
begin
  if RunTimeErrorPresent then Exit;
  if OSMsgBlk^.InPtr = OSMsgBlk^.OutPtr then Exit;
  GSendPacket^[0] := $f0;  {ACK message number}
  ptr MessageStrType (addr(GSendPacket^[1])) :=
OSMsgBlk^.Messages[OSMsgBlk^.OutPtr];
{ CopyPacketToXmtBuff; }
OSMsgBlk^.OutPtr := (OSMsgBlk^.OutPtr + 1) mod 50;
AckNeeded := false;
end;

procedure ServiceProcessEthernetReq;
begin
  if RunTimeErrorPresent then Exit;
  if not QReady(EtherQ) then Exit;
  ExecQ(EtherQ);
  { CopyPacketToXmtBuff; }
  AckNeeded := false;
end;

procedure SendPacket; forward;

procedure EthernetResetCPU;
var
  i : longint;
begin
  GRecvPacket^[0] := $ff;
  if MyCPUId = 1 then
    for i := 1 to CPUCount do
      if CPUNumbers[i] <> 1
        then InterruptProcessor(CPUNumbers[i]);
  ResetCPU;
end;

procedure HandlelOReq;
var idongint;
begin
  if DoMsg AND ServeOSMsg then begin
    PostMessageToEthernet;
    if AckNeeded then ServiceProcessEthernetReq;
  end else begin
    ServiceProcessEthernetReq;
    if AckNeeded then PostMessageToEthernet;
  end;
  ServeOSMsg := NOT ServeOSMsg;
  if RunTimeErrorPresent then begin
    GSendPacket^[0] := $ff;  {error indicator}
    GSendPacket^[1] := $fe;  {run-time error indicator}
    Longint2Packet(ErrorMessagePtr,2);
    move(ErrorMessageBuffer,GSendPacket^[6],ErrorMessagePtr);
    { CopyPacketToXmtBuff; }
    SendPacket;
    for i := 1 to 100000 do
      EthernetResetCPU;
  end;
end;
procedure ReceiveResponse;
begin
  if QReady(EtherQ) then ExecQ(EtherQ);
end;

function PacketStr(Index:longint):string;
var
  i : longint;
  S : string;
begin
  S := HexStrL(Index,4) + ' ';
  for i := Index to Index + 15 do
    S := S + HexStrL(RcvByte(i),2) + ' ';
  S := S + ' ';
  for i := Index to Index + 15 do
    if (RcvByte(i) >= 32) AND (RcvByte(i) <= 126) then
      S := S + chr(RcvByte(i))
    else S := S + chr(250);
  PacketStr := S;
end;

procedure BecomeExclusiveProcessor;
begin
  HoldGlobalBus;
  CpuCount := 1;
  CPUWindows[CpuCount] := MyWindow;
  CPUNumbers[CpuCount] := MyCPUID;
  ExclusiveCPUDeclared := true;
  ExclusiveCPUNumber := MyCPUID;
  ReleaseGlobalBus;
end;

procedure RespondToEthernet;
begin
  InsideEthernet := true;
  if MyWindow = 0 then IndicatePresence;
  AckNeeded := true;
  case RcvByte(0) of
    1 : DecodeLine;
    2 : DecodeLine2;
    $a0 : BecomeExclusiveProcessor;
    $e0 : DoMsg := false;
    $e1 : DoMsg := true;
    $e2 : DoTrace := false;
    $e3 : DoTrace := true;
    $fd : ReceiveResponse;
    $fe : HandleIOReq;
    $ff : EthernetResetCPU;
    else begin
      RunTimeError(1,'Illegal Ethernet Packet : ' + HexStrL(RcvByte(0),2));
      RunTimeError(0,'Receive Page Address : ' + HexStrL(RcvPage,8));
    end;
end;
RunTimeError(0,PacketStr(-12));
RunTimeError(0,PacketStr(4));
RunTimeError(2,PacketStr(20));
end;
end;
if AckNeeded then GSendPacket^[0] := 0;
SendPacket;
InsideEthernet := false;
end;

procedure CopyPacketToGlobRam;
var
  RcvBaseAddr, {4(a5)}
  RcvPageOffs, {8(a5)}
  DestOffs : longint; {12(a5)}
begin
  RcvBaseAddr := addr(RcvBuff);
  RcvPageOffs := RcvPage;
  DestOffs := addr(GRecvPacket^[0]);
inline($206D/$0004/ { MOVE.L 4(A5),A0 }
    $202D/$0008/ { MOVE.L 8(A5),D0 }
    $226D/$000C/ { MOVE.L 12(A5),A1 }
    $223C/$0000/$02ED/ { MOVE.L #749,D1 }
    {L: }
    $32F0/$0000/ { MOVE.W (D0,A0),(A1) + }
    $5480/ { ADDQ.L #2,D0 }
    $0280/$0000/$7FFF/ { ANDI.L #$7FFF,D0 }
    $51C9/$FFF2); { DBF D1,L }
end;

procedure CopyGSendPacketToXmtBuff;
var
  SourceAddr, {4(a5)}
  DestAddr : longint; {8(a5)}
begin
  DestAddr := addr(XmtBuff[0]);
  SourceAddr := addr(GSendPacket^[0]);
inline($206D/$0004/ { MOVE.L 4(A5),A0 }
    $226D/$000C/ { MOVE.L 12(A5),A1 }
    {L: }
    $32D8/ { MOVE.W (A0)+,(A1) + }
    $51C9/$FFF2); { DBF D1,L }
end;

procedure SendPacket; {NOTE: All packets are 1512 bytes long, including addrs}
const

Length = 1512;
var
  DoReset,
  FatalError : boolean;
begin
  CopyGSendPacketToXmtBuff;
  FatalError := GSendPacket^[0] = $ff;
  DoReset := GSendPacket^[0] = $8e;
  EtherCPU[2 * TPSR] := 0; {Packet starts at page zero.}
  EtherCPU[2 * TBCR0] := Length and $ff; {Low byte of count}
  EtherCPU[2 * TBCR1] := (Length div $100) and $ff; {Hi byte of count}
  EtherCPU[2 * CMDR] := $24;
  if FatalError OR DoReset then begin
    Delay(10000);
    EthernetResetCPU;
  end;
end;

procedure EtherInterrupt(FormatVector:word;ProgCntnr:longint;StatusReg:word;
                        a7,a6,a5,a4,a3,a2,a1,a0,
                        d7,d6,d5,d4,d3,d2,d1,d0 : longint);exception;
var
  IntStatus : byte;
  DestinationCPUNum,
  LEDState : longint;
begin
  SetLED(0,LEDState);
  {★★★Determine the cause of the interrupt★★★}
  IntStatus := EtherCPU[2 * ISR];
  {★★★Clear the interrupt★★★}
  EtherCPU[2 * ISR] := $ff;
  {★★★Start checking for possible interrupt causes.★★★}
  {★★★Check for good receive.★★★}
  if (IntStatus and MSK_PRX) <> 0 then begin
    {★★★ Make a copy of the packet for the appropriate processor.★★★}
    CopyPacketToGlobRam;
    DestinationCPUNum := ptr byte (RcvPage - 1 + RcvRam);
    {★★★ Update the RcvPage pointer.★★★}
    RcvPage := $10 +
               $100 * ($FF AND (ptr byte (RcvPage - 15 + RcvRam) - $80 ));
    if DestinationCPUNum = 1 then RespondToEtheremet
    else InterruptProcessor(DestinationCPUNum);
  end;
  SetLED(LEDState,LEDState);
end;

procedure MailBoxInterrupt;exception;
begin
  MonSwt := $20;
  SendPacket;
end;
procedure InitEthernet;
begin
  Delay(6000);
  {***Select Page 0***}
  EtherCPU[2 * CMDR] := MSK_PG0 + MSK_RD2;
  {***Select FIFO threshold = 8 bytes, word-wide DMA***}
  EtherCPU[2 * DCR] := MSK_WTS + MSK_BMS + MSK_FT10;
  {***Clear RBCR0,1***}
  EtherCPU[2 * RBCR0] := 0;
  EtherCPU[2 * RBCR1] := 0;
  {***Disable the receiver***}
  EtherCPU[2 * RCR] := MSK_MON;
  {***Set TCR for normal operation***}
  EtherCPU[2 * TCR] := 0;
  {***Set Stop Page to end of RAM***}
  EtherCPU[2 * PSTOP] := $00;
  {***Set Start Page to $80, reserving first 2k for transmit***}
  EtherCPU[2 * PSTART] := $80;
  {***Select Boundary Register to Unused Page to prevent Overwrite Error.***}
  EtherCPU[2 * BNRY] := $70;
  {***Clear ISR by 1's***}
  EtherCPU[2 * ISR] := $ff;
  {***Enable appropriate interrupts***}
  EtherCPU[2 * IMR] := MSK_PRX + MSK_RXE + MSK_TXE
                    + MSK_OVW + MSK_CNT;
  {***Select Page 1***}
  EtherCPU[2 * CMDR] := MSK_PG1 + MSK_RD2;
  {***Initialize the Physical Address registers***}
  EtherCPU[2 * PAR0] := 0;
  EtherCPU[2 * PAR1] := ord('C');
  EtherCPU[2 * PAR2] := ord('H');
  EtherCPU[2 * PAR3] := ord('E');
  EtherCPU[2 * PAR4] := ord('N');
  EtherCPU[2 * PAR5] := MyCPUID;
  {***Initialize the Multicast Address registers***}
  EtherCPU[2 * MAR0] := $ff;
  EtherCPU[2 * MAR1] := $ff;
  EtherCPU[2 * MAR2] := $ff;
  EtherCPU[2 * MAR3] := $ff;
  EtherCPU[2 * MAR4] := $ff;
  EtherCPU[2 * MAR5] := $ff;
  EtherCPU[2 * MAR6] := $ff;
  EtherCPU[2 * MAR7] := $ff;
  {***Initialize CURR = PSTART + 1 ***}
  EtherCPU[2 * CURR] := $80;
  {***Also initialize the software page pointer***}
  RcvPage := $10;
  RunTimeErrorPresent := false;
  InsideEthernet := false;
  {***Select Page 0***}
  EtherCPU[2 * CMDR] := MSK_PG0 + MSK_RD2;
{***Put the 8390 on line***}
EtherCPU[2 * CMDR] := MSK_STA + MSK_RD2;

{***Program the RCR for normal operation***}
EtherCPU[2 * RCR] := MSK_AB;

{*** Move in the source & destination addresses. ***}
ptr word (addr(XmtBuff[-12])) := $100 * 0 + ord('C');
ptr word (addr(XmtBuff[-10])) := $100 * ord('H') + ord('E');
ptr word (addr(XmtBuff[-8])) := $100 * ord('N') + 0;
ptr word (addr(XmtBuff[-6])) := $100 * 0 + ord('C');
ptr word (addr(XmtBuff[-4])) := $100 * ord('T') + ord('E');
ptr word (addr(XmtBuff[-2])) := $100 * ord('N') + MyCPUID;

{** Prepare for ethernet interrupts. **}
SetExceptionVec(28,@EtherInterrupt);
SetExceptionVec(26,@MailBoxInterrupt);
end;

procedure RunTimeError { (Action:longint;Message:string) } ;
var
i : longint;
begin
if Action AND 1 <> 0 then ErrorMessagePtr := 0;
for i := 0 to ord(Message[0]) do begin
inc(ErrorMessagePtr);
ErrorMessageBuffer[ErrorMessagePtr] := Message[i];
end;
if Action AND 2 = 0 then Exit;
RunTimeErrorPresent := true;
if InsideEthernet then begin
GSendPacket^[0] := $ff;  {error indicator}
GSendPacket^[1] := $fe;  {run-time error indicator}
Longint2Packet(ErrorMessagePtr,2);
move(ErrorMessageBuffer,GSendPacket^[6],ErrorMessagePtr);
SendPacket;
for i := 1 to 100000 do ;
{ ResetCPU; }
end;
Flick(0);
end;

function GetRamPage;
var
MemError:  boolean;
i :  longint;
begin
MemError := false;  {assume no error}
{no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)}
$007c/$0700);  {ori.w #$0700, st}
if FreePagePtr > RamPagesM1 then MemError := true
else begin
i := PageNum[FreePagePtr];
end;
end;
inc(FreePagePtr);
end;
{interrupts back on}
inline($46df); {move.w (a7)+,sr}
if MemError then GetRamPage := GetGlobalPage
else GetRamPage := RamBase + i * PageSize;
end;

procedure FreeRamPage;
var i:longint;
begin
if (Num < RamBase) OR (Num >= (RamBase + RamSize))
then ReleaseGlobalPage(Num)
else begin
i := (Num - RamBase) div PageSize;
{no interrupts allowed}
inline($40e7/ {move.w sr,-(a7)}
$007c/$0700); {ori.w #$0700,st}
dec(FreePagePtr);
PageNum[FreePagePtr] := i;
{interrupts back on}
inline($46df); {move.w (a7)+,sr}
end;
end;

procedure InitMem;
var i:longint;
begin
for i := 0 to RamPagesM1 do PageNum[i] := i;
FreePagePtr := (addr(LastVariable) - RamBase + PageSize - 1) div PageSize;
{ FreePagePtr now points to the first available page. }
CodeHighPtr := UserCodeBase - 1;
NumCodePages := 0;
end;

procedure PlaceOnProcessor(ProcessorNum,PPage : longint);
var A : longint;
begin
A := CPUWindows[ProcessorNum];
HoldGlobalBus;
inc(ptr WaitingPType(A).WaitCount);
ptr WaitingPType(A).WaitingProcs[ptr WaitingPType(A).WaitCount] := PPage;
ReleaseGlobalBus;
end;

procedure CopyPParms(PIAddr,PPage,ParmAddrJ Directiondongint);
var
NumParms,
ParmType,ParmSize,
PtrThere,
i : longint;
PDPtr : PProcessData;
begin
PDPtr := PDataPtr(PPage);
PtrThere := PDPtr^ .ProcessParamAddr;
NumParms := WReadUP(PIAddr+10);
for i := 0 to NumParms-1 do begin
ParmType := WReadUP(PIAddr + 12 + i*6) and $ff;
ParmSize := LReadUP(PIAddr + 14 + i*6);
case ParmType of
  0 : begin {val parameter, value on stack}
      if Direction=1 then
        Proc2ProcCopy(addr(i),ParmAddr,PPage,PtrThere,ParmSize);
      ParmAddr := (ParmAddr + ParmSize + 1) and $fffffffe;
      PtrThere := (PtrThere + ParmSize + 1) and $fffffffe;
      end;
  1 : begin {val parameter, address on stack}
      if Direction=1 then
        Proc2ProcCopy(addr(i),LReadUD(ParmAddr),PPage,PtrThere,ParmSize);
      inc(ParmAddr,4);
      PtrThere := (PtrThere + ParmSize + 1) and $fffffffe;
      end;
  2 : begin {var parameter, address on stack}
      if Direction=1 then
        Proc2ProcCopy(addr(i),LReadUD(ParmAddr),PPage,PtrThere,ParmSize)
      else
        Proc2ProcCopy(PPage,PtrThere,addr(i),LReadUD(ParmAddr),ParmSize);
      inc(ParmAddr,4);
     PtrThere := (PtrThere + ParmSize + 1) and $fffffffe;
      end;
  3 : begin {ret parameter, address on stack}
      if Direction=0 then
        Proc2ProcCopy(PPage,PtrThere,addr(i),LReadUD(ParmAddr),ParmSize);
      inc(ParmAddr,4);
      PtrThere := (PtrThere + ParmSize + 1) and $fffffffe;
      end;
  end;
end;
const
MaxStatProcess = 64;

var
ChanRsrcCount : longint;
ProcData = array[1..MaxStatProcess] of record
  PAddr,
  RCount : longint;
end;

procedure ScanStructParam(var ParmData : longint;
ParmAddr : longint;

var
  Count,
  Stride,
  TempData,
  PType : longint;
begin
  PType := WReadUP(ParmData);
  inc(ParmData,2);
  case PType of
    1 : inc(ChanRsrcCount);
    2 : inc(ChanRsrcCount);
    9 : begin {ARRAY}
      Count := LReadUP(ParmData);
      inc(ParmData,4);
      Stride := LReadUP(ParmData);
      inc(ParmData,4);
      while Count > 0 do begin
        dec(Count);
        TempData := ParmData;
        ScanStructParam(TempData,ParmAddr);
        inc(ParmAddr,Stride);
      end;
      ParmData := TempData;
    end;
    10 : begin {RECORD}
      Count := WReadUP(ParmData);
      inc(ParmData,2);
      while Count > 0 do begin
        dec(Count);
        Stride := LReadUP(ParmData);
        inc(ParmData,4);
        ScanStructParam(ParmData,ParmAddr+Stride);
      end;
    end;
  end;
end;

procedure TestProcessParms(PVarAddr: longint);
var
  i,
  SDataAddr,
  NumParam,
  ParamType,
  CodeAddr : longint;
begin
  CodeAddr := LReadUD(PVarAddr+6);
  NumParam := WReadUP(CodeAddr+10);
  inc(CodeAddr,12);  { point to first parameter's data }
  inc(PVarAddr,12);  { point to first parameter's address }
  SDataAddr := CodeAddr + 6 * NumParam;  { points to data for arrays }
for i := 1 to NumParam do begin
  ParamType := WReadUP(CodeAddr);
  case (ParamType AND $00ff) of
    2, 3 : inc(PVarAddr, 4);
    0 : PVarAddr := (PVarAddr + LReadUP(CodeAddr + 2)) AND $ffffff0;
    1 : begin
      case (ParamType AND $ff00) of
        $0000 : ScanStructParam(SDataAddr, LReadUD(PVarAddr));
        $0100 : inc(RsrcChanCount);
        $0200 : inc(RsrcChanCount);
        end;
      end;
      inc(PVarAddr, 4);
    end;
  end;
inc(CodeAddr, 6);
end;
end;

procedure StoreProcAssignments(PCount, PVarAddr: longint);
var
  RPerProc,
  RThisProc,
  PCurr,
  i: longint;
begin
  PCurr := 1;
  RPerProc := RsrcChanCount div CPUCount div 2;
  RThisProc := 0;
  for i := 1 to PCount do begin
    if i <= MaxStatProc then begin
      WWriteUD(ProcData[i].PAddr + 10, PCurr);
      inc(RThisProc, ProcData[i].RCount);
      if RThisProc >= RPerProc then begin
        PCurr := 1 + (PCurr mod CPUCount);
        RThisProc := 0;
      end;
    end else begin
      WWriteUD(PVarAddr + 10, PCurr);
      PCurr := 1 + (PCurr mod CPUCount);
    end;
    PVarAddr := LReadUD(PVarAddr + 2);
  end;
end;

procedure DetermineProcessMapping(PCount, PVarAddr: longint);
var
  i,
  LocalCount,
  TAddr: longint;
begin
  TAddr := PVarAddr;
end;
RsrcChanCount := 0;
for i := 1 to PCount do begin
    {*** Count the number of resource and channel parameters ***}
    LocalCount := RsrcChanCount;
    TestProcessParms(TAddr);
    LocalCount := RsrcChanCount - LocalCount;
    {*** Store the result along with the process address ***}
    if i <= MaxStatProc then begin
        ProcData[i].PAddr := TAddr;
        ProcData[i].RCount := LocalCount;
    end;
    TAddr := LReadUD(TAddr + 2);
end;
StoreProcAssignments(PCount, PVarAddr);
end;

procedure DoParallelCall(FirstProcessVarAddr:longint);
var
    ParentID,
    PPage,
    ProcessVarAddr,
    ProcessCount : longint;
    CurrentCount,
    Update,
    ChildPage,
    ChildMMU,
    i,
    PAddr,
    RelPage,
    StatPage,
    UpdateVal : longint;
    PDPtr : PProcessData;
    LDPtr : ^LevelDDescTable;
begin
    ProcessVarAddr := FirstProcessVarAddr;

    if CPUCount > 1 then begin

        {*** Count the processes. ***}
        ProcessCount := 0;
        while PAddr <> -1 do begin
            inc(ProcessCount);
            PAddr := LReadUD(PAddr + 2);
        end;

        {* Determine process mapping. *}
        DetermineProcessMapping(ProcessCount, FirstProcessVarAddr);
    end else begin

        {* If only 1 CPU exists, place each Process on CPU #1 *}

PAddr := FirstProcessVarAddr;
while PAddr <> -1 do begin
    WWriteUD(PAddr + 10, 1);
PAddr := LReadUD(PAddr + 2);
end;

{* Get a page for finished processes to respond in. *}
ParentID := GetGlobalPage;
ptr WaitingPType(ParentID).WaitCount := 0;

ProcessCount := 0;
while ProcessVarAddr <> -1 do begin
    PPage := CreateProcessSpace(LReadUD(ProcessVarAddr+6),1);
    PDPtr := PDataPtr(PPage);
    PDPtr^.ProcessParentID := ParentID;
    PDPtr^.ProcessParentAddr := ProcessVarAddr;
    CopyParms(LReadUD(ProcessVarAddr+6),PPage,ProcessVarAddr+12,1);
    PlaceOnProcessor(WReadUD(ProcessVarAddr + 10),PPage);
    inc(ProcessCount);
    ProcessVarAddr := LReadUD(ProcessVarAddr + 2);
end;

if ProcessCount <> 0 then begin
    LeaveQ(ReadyQ);
    JoinQFlick(ParentProcQ,ParentID);
    while ProcessCount > 0 do begin
        while ptr WaitingPType(ParentID).WaitCount > 0 do begin
            HoldGlobalBus;
            CurrentCount := ptr WaitingPType(ParentID).WaitCount;
            ChildPage := ptr WaitingPType(ParentID).WaitingProc[CurrentCount];
            dec(ptr WaitingPType(ParentID).WaitCount);
            ReleaseGlobalBus;
            {* Get the address of the child's process variable. *}
            PDPtr := PDataPtr(ChildPage);
            ProcessVarAddr := PDPtr^.ProcessParentAddr;
            {* Copy back any VAR or RET parameters. *}
            CopyParms(LReadUD(ProcessVarAddr+6),ChildPage,ProcessVarAddr+12,0);
            {* Mark the process as available. *}
            WWriteUD(ProcessVarAddr,0);
            {* Decrement the count towards zero. *}
            dec(ProcessCount);
            {* Release all global pages assigned to the process. *}
            {* First, make sure the process is done. *}
            repeat until PDPtr^.PPageBusy = 0;
            {* Free any pages used in the mmu. *}
            ptr longint(addr(LDPtr)) := PDPtr^.MmuPages[UserDataLevelCIndex] AND (-PageSize);
            for i := 0 to 1023 do
if LDLtr^ [i] <> 0 then ReleaseGlobalPage(LDPtr^ [i]);
{ * Release the Mmu page and the process page. *}
ReleaseGlobalPage(PDPtr^ .MmuPages[UserDataLevelCIndex]);
ReleaseGlobalPage(ChildPage);
end;
if ProcessCount <> 0 then begin
LeaveQ(ParentProcQ); {leave the parent process queue}
JoinQFlick(ParentProcQ,ParentID); {join it again at the end}
end;
end;
FreeRamPage(ParentID);
LeaveQ(ParentProcQ);
JoinQFlick(ReadyQ,0);
end else Relinquish;
end;

procedure ReturnFromProcess(CodeAddress:longint);
var
  ParentID,
  ThisPPage : longint;
  PDPtr : PProcessData;
begin
  {* Get page address for this process, parent process. *}
  ThisPPage := addr(ThisPPage) and (-PageSize);
  PDPtr := PDataPtr(ThisPPage);
  ParentID := PDPtr^ .ProcessParentID;
{ * Discard any Local ram pages, remove yourself from queue. }
RemoveLocalPages(ThisPPage);

  {* Mark the process page as busy - Flick will clear this *}
PDPtr^ .PPageBusy := -1;

  {* Signal the parent process that you are done. *}
  HoldGlobalBus;
  inc(ptr WaitingPType(ParentID).WaitCount);
  ptr WaitingPType(ParentID).WaitingProcs[ptr WaitingPType(ParentID).WaitCount] := ThisPPage;
  ReleaseGlobalBus;
  {* Signal parent processor. *}

  {* Relinquish control to the CPU. *}
  LeaveQ(ReadyQ);
  Flick(0);
end;

procedure CheckForNewProcesses;
var
  PPage : longint;
begin
  while MyWait^ .WaitCount <> 0 do begin

HoldGlobalBus;
PPage := MyWait^.WaitingProcs[MyWait^.WaitCount];
dec(MyWait^.WaitCount);
ReleaseGlobalBus;
InitLocalProcess(PPage);
PutOnQ(ReadyQ,PPage);
end;
end;

procedure CheckForFinishedProcesses;
var
  i : longint;
begin
  for i := 1 to ParentProcQ.PCount do ExecQ(ParentProcQ);
end;

type
  ResourceWaitType = record
    ResourcePage,
    MyWaitNumber : longint;
  end;

function InitializeResource(StackSize,VarSize:longint):longint;
var
  i,
  RamPageCount,
  RPage : longint;
  PDPtr : PProcessData;
  LDPtr : ^LevelDDescTable;
begin
  (* Get Pages needed for process page, mmu page. *)
  RPage := GetGlobalPage;
  PDPtr := PDataPtr(RPage);
  for i := 0 to 7 do PDPtr^.MmuPages[i] := 0;
  ptr longint (addr(LDPtr)) := GetGlobalPage;
  PDPtr^.MmuPages[UserDataLevelCIndex] := ptr longint (addr(LDPtr)) + 2;
  (* Clear the MMU Page *)
  for i := 0 to 1023 do LDPtr^[i] := 0;
  RamPageCount := (StackSize + VarSize + PageSize - 1) div PageSize;
  for i := 0 to RamPageCount - 1 do LDPtr^[i] := GetGlobalPage + 1;
  (* Indicate that the MMU will be used. *)
  PDPtr^.UseMMU := 1;
  (* Clear the IO Buffer for the resource. *)
  PDPtr^.IOBuffer[0] := #0;
  PDPtr^.NextSelectNum := 0;
PDPtr^.ResourceNext2Exec := 0; {next wait allowed in}
PDPtr^.ResourceNextWait := 0; {next wait number to take}
PDPtr^.ResourceEndParms := StackSize + UserDataBase;
  InitializeResource := RPage;
end;
procedure DiscardResource(RPage:longint);
var
  i : longint;
  PDPtr : PProcessData;
  LDPtr : LevelDDescTable;
begin
  PDPtr := PDataPtr(RPage);
  ptr longint (addr(LDPtr)) := PDPtr^.MMUPages[UserDataLevelCIndex]
    AND (-PageSize);
  for i := 0 to 1023 do
    if LDPtr^[i] <> 0 then FreeRamPage(LDPtr^[i]);
  FreeRamPage(PDPtr^.MmuPages[UserDataLevelCIndex]);
  FreeRamPage(RPage);
end;

function TakeANumber(RPage:longint):longint;
var
  PDPtr : PProcessData;
begin
  PDPtr := PDataPtr(RPage);  
  HoldGlobalBus;  
  TakeANumber := PDPtr^.ResourceNextWait;  
  inc(PDPtr^.ResourceNextWait);  
  ReleaseGlobalBus;
end;

function ResourceBusy(var RW ait: ResourceWaitType):boolean;
var
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr (RWait.ResourcePage);
end;

procedure CallResourceProc(CodeAddr:longint;var ParamAddr : longint);
var
  TmpVar,
  SearchAddr,
  i,
  CodeAddr2,
  ParamCount,
  ParamSize,
  RsrcParamAddr,
  RPage : longint;
  MyLocalWait : ResourceWaitType;
  Found : boolean;
  PDPtr : PProcessData;

procedure CodeW(w:word);
begin
  ptr word(CodeAddr2) := w;
CodeAddr2 := CodeAddr2 + 2;
end;

procedure CodeL(l:longint);
beginn
  ptr longint(CodeAddr2) := l;
  CodeAddr2 := CodeAddr2 + 4;
end;

begin
  RPage := LReadUD(ParamAddr); inc(ParamAddr,4);
  MyLocalWait.ResourcePage := RPage;
  MyLocalWait.MyWaitNumber := TakeANumber(RPage);
  if ResourceBusy(MyLocalWait) then begin
    LeaveQ(ReadyQ);
    JoinQFlick(ResourceWaitQ,addr(MyLocalWait));
    { When the resource becomes available, we will be placed back onto
      the ReadyQ.}
  end;
  ParamCount := WReadUP(CodeAddr+10);
  ParamSize := 0;
  for i := 0 to ParamCount - 1 do begin
    ParamSize := ParamSize + LReadUP(CodeAddr + 14 + i*6);
    ParamSize := (ParamSize + 1) and $ffffff;
  end;
  PDPtr := PDataPtr (RPage);
  RsrcParamAddr := PDPtr ^ .ResourceEndParms - ParamSize;
  PDPtr ^ .ProcessParamAddr := RsrcParamAddr;
  {* Copy the parameters into the resource's address space.}
  CopyPParms(CodeAddr,RPage,ParamAddr,1);

  {* Poke in machine instructions into the resource's process page.}
  First, these instructions will save the calling program's a7.
  Then, the new value of a7 is loaded into a7.
  Then, load the appropriate values into a4,a6, and usp.
  Next, they will build up an exception stack frame.
  Finally, they will rte into the process.}

  CodeAddr2 := RPage;
  CodeW($23cf);  {move.l a7,RPage+ResourceParentA7}
  CodeL(addr(PDPtr ^ .ResourceParentA7));
  CodeW($2e7c);  {movea.l #(A5Value),a7}
  CodeL(addr(PDPtr ^ .A5Value));
  CodeW($287c);  {movea.l #RsrcParamAddr,a4}
  CodeL(RsrcParamAddr);
  CodeW($4e64);  {move.l a4,usp}
  CodeW($2c7c);  {movea.l #EndParms,a6}
  CodeL(PDPtr ^ .ResourceEndParms);
  CodeW($3f3c0000);  {move.w #0,-(a7)} {frame/vector}
  CodeW($2f3c);  {move.l #ProgAddr,-(a7);ret addr}
CodeL(CodeAddr);
CodeW($3f3c);
if DoTrace then
    CodeW($9000 + $100 * ProcessIntLevel)
        {move.w #$9000,-(a7)} {status reg for trace}
else
    CodeW($1000 + $100 * ProcessIntLevel);
        {move.w #$1000,-(a7)} {status reg}
CodeW($4e73); {rte}

{* Before jumping into the resource procedure, we must remove the current
process from the ready queue, inserting the resource procedure process
into the queue at the exact same position. After the resource procedure
is completed, we will reverse the procedure.*}

SwapProcessOnQ(addr(TmpVar) and (-PageSize),RPage);

RestoreMmu(RPage); {Set up the MMU for the resource to be called.}

TmpVar := RPage; {We'll jsr to this address}

TimerStart;
inline($2f0c/ {move.l a4,-(a7)}
    $2f0d/ {move.l a5,-(a7)}
    $2f0e/ {move.l a6,-(a7)}
    $206d/$0004/ {movea.l 4(a5),a0}
    $4e90/ {jsr (a0)}
    $2c5f/ {movea.l (a7)+,a6}
    $2a5f/ {movea.l (a7)+,a5}
    $285f); {movea.l (a7)+,a4}

TimerStop;

{* Swap the processes back to restore the ReadyQ*}
SwapProcessOnQ(RPage,addr(TmpVar) and (-PageSize));

{Set up the MMU for the process being returned to.}
RestoreMmu(addr(i) and (-PageSize));

{* Copy the parameters back into this process' address space.*}
CopyPParms(CodeAddr,RPage,ParamAddr, 0);

{* Adjust the stack back in the calling process - remove params.*}
for i := 0 to ParamCount - 1 do
    case WReadUP(CodeAddr + 12 + i*6) and $ff of
    0 : ParamAddr := (ParamAddr + LReadUP(CodeAddr + 14 + i*6) + 1) and $fffffff;
        1,2,3 : ParamAddr := ParamAddr + 4;
end;
{* Indicate that we are done with the resource. *}
inc(PDPtr^.ResourceNext2Exec);

{* Leave the Ready Queue, then add process back onto end of Ready Queue. *}
Relinquish;
end;

procedure ReturnFromResourceProc;
var
  RtsStackPtr: longint;
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr(addr(PDPtr));
  RtsStackPtr := PDPtr^.ResourceParentA7;
  inline($2e6d/$0004/ {movea.l 4(a5),a7}
        $4e75); {rts}
end;

procedure CheckForReadyResources;
var
  SearchAddr,
  TmpAddr,
  i: longint;
  PDPtr: PProcessData;
begin
  SearchAddr := GetFirstProcess(ResourceWaitQ);
  while SearchAddr <> -1 do begin
    PDPtr := PDataPtr(SearchAddr);
    i := PDPtr^.QDataPtr;
    if not ResourceBusy(ptr ResourceWaitType(i)) then begin
      TmpAddr := GetNextProcess(ResourceWaitQ,SearchAddr);
      TakeOffQ(ResourceWaitQ,SearchAddr);
      PutOnQ(ReadyQ,SearchAddr);
      SearchAddr := TmpAddr;
    end else SearchAddr := GetNextProcess(ResourceWaitQ,SearchAddr);
  end;
end;

const
  NearDataSize = 4096 - (7 + 64) * 4;

type
  GChanType = record
    ReadStat,
    WriteStat,
    OpenStat,
    EOCStat,
    ReadProc,
    WriteProc,
    DataSize: longint;
    FarPtrs: array[0..63] of longint;
  end;
NearData : array[0..NearDataSize-1] of byte;
end;

function InitChannel(DSize:longint):longint;
var
  CPage : longint;
  GC : ^GChanType absolute CPage;
  i : longint;
begin
  CPage := GetGlobalPage;
  GC^.DataSize := DSize;
  GC^.ReadStat := 0;
  GC^.WriteStat := 0;
  GC^.OpenStat := 0;
  GC^.EOCStat := 0;
  i := 0;
  while DSize > NearDataSize do begin
    GC^.FarPtrs[i] := GetGlobalPage;
    DSize := DSize - PageSize;
    inc(i);
  end;
  InitChannel := CPage;
end;

procedure DelInitChannel(ChanAddr:longint);
var
  GC : ^GChanType absolute ChanAddr;
  i : longint;
begin
  i := 0;
  while GC^.DataSize > NearDataSize do begin
    FreeRamPage(GC^.FarPtrs[i]);
    GC^.DataSize := GC^.DataSize - PageSize;
    inc(i);
  end;
  FreeRamPage(ChanAddr);
end;

function ChkEOC(ChanAddr:longint):longint;
var
  GC : ^GChanType absolute ChanAddr;
begin
  ChkEOC := 1;
  if (GC^.ReadProc = MyProcID) AND ((GC^.OpenStat AND $30) = $30) then ChkEOC := 0;
end;

procedure ResetChan(ChanAddr:longint;CodeAddr:longint);
var
  GC : ^GChanType absolute ChanAddr;
begin
procedure RewriteChan(ChanAddr:longint;CodeAddr:longint);
var
    GC : ^GChanType absolute ChanAddr;
    OldVal,NewVal,i : longint;
begin
    HoldGlobalBus;
    if (GC^.OpenStat AND $11) <> 0 then begin
        ReleaseGlobalBus;
        RunTimeError(3,'Error Rewriting Channel at Address $'
                     + HexStrL(CodeAddr,8));
    end;
    GC^.OpenStat := GC^.OpenStat OR $01;
    ReleaseGlobalBus;
    GC^.WriteProc := MyProcID;
    HoldGlobalBus;
    ReleaseGlobalBus;

    if (GC^.OpenStat AND $10) = 0 then begin
        LeaveQ(ReadyQ);
        JoinQFlick(OpenChanQ,ChanAddr);
    end else begin
        if (GC^.WriteProc AND $ff) = MyCPUID then begin
            TakeOffQ(OpenChanQ,GC^.WriteProc);
            PutOnQ(ReadyQ,GC^.WriteProc);
        end else begin
            { Signal other processor. }
        end;
        Relinquish;
    end;
end;

HoldGlobalBus;
if (GC^.OpenStat AND $22) <> 0 then begin
    ReleaseGlobalBus;
    RunTimeError(3,'Error Resetting Channel at Address $'
                 + HexStrL(CodeAddr,8));
end;
GC^.OpenStat := GC^.OpenStat OR $02;
ReleaseGlobalBus;
GC^.ReadProc := MyProcID;
HoldGlobalBus;
ReleaseGlobalBus;

if (GC^.OpenStat AND $10) = 0 then begin
    LeaveQ(ReadyQ);
    JoinQFlick(OpenChanQ,ChanAddr);
end else begin
    if (GC^.WriteProc AND $ff) = MyCPUID then begin
        TakeOffQ(OpenChanQ,GC^.WriteProc);
        PutOnQ(ReadyQ,GC^.WriteProc);
    end else begin
        { Signal other processor. }
    end;
    Relinquish;
end;

procedure RewriteChan(ChanAddr:longint;CodeAddr:longint);
var
    GC : ^GChanType absolute ChanAddr;
    OldVal,NewVal,i : longint;
begin
    HoldGlobalBus;
    if (GC^.OpenStat AND $11) <> 0 then begin
        ReleaseGlobalBus;
        RunTimeError(3,'Error Rewriting Channel at Address $'
                     + HexStrL(CodeAddr,8));
    end;
    GC^.OpenStat := GC^.OpenStat OR $01;
    ReleaseGlobalBus;
    GC^.WriteProc := MyProcID;
    HoldGlobalBus;
    ReleaseGlobalBus;

    if (GC^.OpenStat AND $20) = 0 then begin
        LeaveQ(ReadyQ);
        JoinQFlick(OpenChanQ,ChanAddr);
    end else begin
        if (GC^.WriteProc AND $ff) = MyCPUID then begin
            TakeOffQ(OpenChanQ,GC^.WriteProc);
            PutOnQ(ReadyQ,GC^.WriteProc);
        end else begin
            { Signal other processor. }
        end;
        Relinquish;
    end;
end;
if (GC^.ReadProc AND $ff) = MyCPUID then begin
  TakeOffQ(OpenChanQ,GC^.ReadProc);
  PutOnQ(ReadyQ,GC^.ReadProc);
end else begin
  { Signal other processor. }
end;
Relinquish;
end;
end;

procedure ReadChan(ChanAddr,DataAddr,DataSize:longint;CodeAddr:longint);
var
  GC : ^GChanType absolute ChanAddr;
  i : longint;
begin
  if GC^.ReadProc <> MyProcID
    then RunTimeError(3,'Error Reading Channel at Address $'
                      + HexStrL(CodeAddr,8));
  if (GC^.OpenStat AND $30) <> $30 then
    RunTimeError(3,'Error Reading Channel at Address $'
                      + HexStrL(CodeAddr,8));
  if GC^.WriteStat = 0 then begin
    GC^.ReadStat := -1;
    LeaveQ(ReadyQ);
    JoinQFlick(RcvChanQ,ChanAddr);
    GC^.ReadStat := 0;
  end;
  { When we get here, GC^.WriteStat <> 0 }
  { Now, copy the data from the channel into the user space. }
  if DataSize > NearDataSize then begin
    BlockUDWrite(addr(GC^.NearData),DataAddr,NearDataSize);
    inc(DataAddr,NearDataSize);
    dec(DataSize,NearDataSize);
  end else begin
    BlockUDWrite(addr(GC^.NearData),DataAddr,DataSize);
    DataSize := 0;
  end;
  i := 0;
  while DataSize > 0 do begin
    if DataSize > PageSize then begin
      BlockUDWrite(GC^.FarPtrs[i],DataAddr,PageSize);
      inc(DataAddr,PageSize);
      dec(DataSize,PageSize);
    end else begin
      BlockUDWrite(GC^.FarPtrs[i],DataAddr,DataSize);
      DataSize := 0;
    end;
    inc(i);
  end;
  { Indicate that the data has been read. }
  GC^.WriteStat := 0;
{ See if we should close the channel. }
if GC^.EOCStat <> 0 then begin
    GC^.OpenStat := 0;
    GC^.EOCStat := 0;
end;
{ If possible, remove the sending process from SendChanQ. }
if (GC^.WriteProc AND $ff) = MyCPUID then begin
    TakeOffQ(SendChanQ,GC^.WriteProc);
    PutOnQ(ReadyQ,GC^.WriteProc);
end else begin
    { Signal other processor. }
    end;
    { Let other processes execute. }
    Relinquish;
end;

procedure WriteChan(ChanAddr,DataAddr,DataSize:longint;CodeAddr:longint;
DoClose:boolean);
var
    GC : ^GChanType absolute ChanAddr;
    i : longint;
begin
    if GC^.WriteProc <> MyProcID
        then RunTimeError(3,'Error Writing Channel at Address $' +
            HexStrL(CodeAddr,8));
    if (GC^.OpenStat AND $30) <> $30 then
        RunTimeError(3,'Error Writing Channel at Address $' +
            HexStrL(CodeAddr,8));
    { Copy the data from the user space into the channel. }
    if DataSize > NearDataSize then begin
        BlockUDRead(addr(GC^.NearData),DataAddr,NearDataSize);
        inc(DataAddr,NearDataSize);
        dec(DataSize,NearDataSize);
    end else begin
        BlockUDRead(addr(GC^.NearData),DataAddr,DataSize);
        DataSize := 0;
    end;
    i := 0;
    while DataSize > 0 do begin
        if DataSize > PageSize then begin
            BlockUDRead(GC^.FarPtrs[i],DataAddr,PageSize);
            inc(DataAddr,PageSize);
            dec(DataSize,PageSize);
        end else begin
            BlockUDRead(GC^.FarPtrs[i],DataAddr,DataSize);
            DataSize := 0;
        end;
        inc(i);
    end;
    { Indicate that the data is present. }
    if DoClose then GC^.EOCStat := -1 else GC^.EOCStat := 0;
GC^.WriteStat := -1;
{ If possible, place the receiver onto the ReadyQ.}
if (GC^.ReadStat <> 0) then begin
  if (GC^.ReadProc AND $ff) = MyCPUID then begin
    TakeOffQ(RcvChanQ,GC^.ReadProc);
    PutOnQ(ReadyQ, GC^.ReadProc);
  end else begin
    { Signal other processor. }
  end;
end;
{ Now, join the SendChanQ and wait for the receiver to receive. }
LeaveQ(ReadyQ);
JoinQFlick(SendChanQ,ChanAddr);
end;

function CheckChannelStatus(ChanAddr:longint):longint;
var
  GC : ^GChanType absolute ChanAddr;
  i : longint;
begin
  if (GC^.OpenStat AND $30) <> $30 then begin
    CheckChannelStatus := 0;
    exit;
  end;
  if (GC^.ReadProc = MyProcID) AND (GC^.WriteStat <> 0) then begin
    CheckChannelStatus := 1;
    exit;
  end;
  if (GC^.WriteProc = MyProcID) AND (GC^.ReadStat <> 0) then begin
    CheckChannelStatus := 1;
    exit;
  end;
  CheckChannelStatus := 0;
end;

procedure ChooseSelectStatement(var StackPtr:longint);
var
  SelectNum,
  P,
  ValidCount : longint;
  PDPtr : PProcessData;
begin
  P := StackPtr + 4;
  ValidCount := 0;
  while LReadUD(P) <> -1 do begin
    P := P + SizeOf(longint);
    inc(ValidCount);
  end;
if ValidCount > 0 then begin
  PDPtr := PDataPtr(addr(PDPtr));
  SelectNum := PDPtr^.NextSelectNum;
  inc(SelectNum);
  PDPtr^.NextSelectNum := SelectNum;
  SelectNum := SelectNum mod ValidCount;
  LWriteUD(P,LReadUD(StackPtr + sizeof(int) * (SelectNum + 1)));
end else begin
  LWriteUD(P,LReadUD(StackPtr));
end;
StackPtr := P;
Relinquish;
end;

procedure CheckForReadyChannelOperations;
var
  SearchAddr,
  TmpAddr,
  ChanAddr : longint;
  GC : ^GChanType absolute ChanAddr;
  PDPtr : PProcessData;
begin
  { Check for completed OPEN operations. }
  SearchAddr := GetFirstProcess(OpenChanQ);
  while SearchAddr <> -1 do begin
    TmpAddr := GetNextProcess(OpenChanQ, SearchAddr);
    PDPtr := PDataPtr(SearchAddr);
    ChanAddr := PDPtr^.QDataPtr;
    if GC^.ReadProc = MyCPUID + SearchAddr then begin
      if (GC^.OpenStat AND $10) <> 0 then begin
        TakeOffQ(OpenChanQ, SearchAddr);
        PutOnQ(ReadyQ, SearchAddr);
      end;
    end else begin
      if (GC^.WriteProc = MyCPUID + SearchAddr)
        AND ((GC^.OpenStat AND $20) <> 0) then begin
        TakeOffQ(OpenChanQ, SearchAddr);
        PutOnQ(ReadyQ, SearchAddr);
      end;
    end;
    SearchAddr := TmpAddr;
  end;

  { Check for completed receive operations. }
  SearchAddr := GetFirstProcess(RcvChanQ);
  while SearchAddr <> -1 do begin
    TmpAddr := GetNextProcess(RcvChanQ, SearchAddr);
    PDPtr := PDataPtr(SearchAddr);
    ChanAddr := PDPtr^.QDataPtr;
    if GC^.WriteStat <> 0 then begin
      TmpAddr := GetNextProcess(RcvChanQ, SearchAddr);
    end;
end;
TakeOffQ(RcvChanQ,SearchAddr);
PutOnQ(ReadyQ,SearchAddr);
end;
SearchAddr := TmpAddr;
end;

{ Check for completed write operations. }
SearchAddr := GetFirstProcess(SendChanQ);
while SearchAddr <> -1 do begin
  TmpAddr := GetNextProcess(SendChanQ,SearchAddr);
  PDPtr := PDataPtr(SearchAddr);
  ChanAddr := PDPtr^.QDataPtr;
  if GC^.WriteStat = 0 then begin
    TakeOffQ(SendChanQ,SearchAddr);
    PutOnQ(ReadyQ,SearchAddr);
  end;
  SearchAddr := TmpAddr;
end;
end;

type
FileType = record
  Status : byte; {0 = closed, 1 = read, 2 = write, 3 = assigned}
  Handle : longint;
  ElementSize : longint;
  FileName : string; {max length of 79}
end;

procedure TRAP2(SuperString:string;
FormatVector:word;ProgCntr:longint;StatusReg:word;
37,36,35,04,33,32,31,80,
d7,d6,d5,d4,d3,d2,dl,d0 : longint) exception;
var alias1 : longint;
alias2 : array[0..3] of byte absolute alias1;
alias3 : ^string absolute a0;
TextFile : ^FileType absolute a1;
TypedFile : ^FileType absolute a0;
i : longint;
alias4 : array[0..7] of byte absolute SuperString;
alias5 : ^longint absolute d1;
UserProg : boolean;
ThisPage,
ProcessAddr,
SmallCount,
DataCount : longint;

procedure StdPckt(Stat,ErrNum:longint);
begin
  GSendPacket^[0] := d0;
  if a1 = -1 then Longint2Packet(-1,1)
else begin
if UserProg then begin
  if BReadUD(addr(ptr FileType(a1).Status)) <> Stat
    then RunTimeError(3,'File Access Error at Address $'
                   + HexStrL(ProgCnt,8));
  Longint2Packet(LReadUD(addr(ptr FileType(a1).Handle)),1);
end else begin
  if TextFile^.Status <> Stat then
    RunTimeError(3,'File Access Error at Address $'
                     + HexStrL(ProgCnt,8));
  Longint2Packet(TextFile^.Handle,1);
end;
end;

procedure FlushlOBuffer;
var FileAddr,FHandle: longint;
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr(ThisPage);
  FileAddr := PDPtr^.CurrentHandle;
  if FileAddr=-1 then FHandle := -1
  else begin
    if UserProg then begin
      if BReadUD(addr(ptr FileType(FileAddr).Status)) <> 2
        then begin
            RunTimeError(1,'File Not Open For Write');
            RunTimeError(2,'User Program Executing At PC = $'
                           + HexStrL(ProgCnt,8));
        end;
      FHandle := LReadUD(addr(ptr FileType(FileAddr).Handle));
    end else begin
      if ptr FileType(FileAddr).Status <> 2 then begin
        RunTimeError(1,'File Not Open For Write');
        RunTimeError(2,'Supervisor Program Executing At PC = $'
                      + HexStrL(ProgCnt,8));
      end;
      FHandle := ptr FileType(FileAddr).Handle;
    end;
    FHandle := ptr FileType(FileAddr).Handle;
  end;
end;
LeaveQ(ReadyQ);
JoinQFlick(EtherQ,0);
GSendPacket^[0] := $17; {print string}
Longint2Packet(FHandle,1);
Longint2Packet(-1,5);
move(PDPtr^.IOBuffer,GSendPacket^[9],256);
PDPtr^.IOBuffer[0] := #0;
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;

procedure ChkForFlush;
var
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr(ThisPage);
  if (PDPtr^.IOBuffer[0] <> #0) AND
      (a1 <> PDPtr^.CurrentHandle) then FlushIOBuffer;
  PDPtr^.CurrentHandle := a1;
end;

procedure AddStr(S:string);
var
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr(ThisPage);
  PDPtr^.IOBuffer := PDPtr^.IOBuffer + S;
end;

procedure AddChar(C:char);
var
  PDPtr: PProcessData;
begin
  PDPtr := PDataPtr(ThisPage);
  if c=#13 then begin
    if ord(PDPtr^.IOBuffer[0]) >= 254 then
      PDPtr^.IOBuffer[0] := #253;
    PDPtr^.IOBuffer := PDPtr^.IOBuffer + C;
    Exit;
  end;
  if c=#10 then begin
    if ord(PDPtr^.IOBuffer[0]) >= 255 then
      PDPtr^.IOBuffer[0] := #255;
    PDPtr^.IOBuffer := PDPtr^.IOBuffer + C;
    Exit;
  end;
  PDPtr^.IOBuffer := PDPtr^.IOBuffer + C;
end;

procedure PrintChar(c,w:longint);
begin
  if w > 127 then w := 127;
  while w > 1 do begin
    AddChar(' ');
    dec(w);
  end;
  AddChar(chr(c));
end;

procedure PrintInt(i,w:longint);
var s:string;
  Negate : boolean;
begin
  if w > 127 then w := 127;
  while w > 1 do begin
    AddChar(' ');
    dec(w);
  end;
  AddChar(chr(c));
end;
if i < 0 then begin
    Negate := true;
i := -i;
end else Negate := false;
s := ""
if i = 0 then s := '0'
else while i <> 0 do begin
    s := chr(i mod 10 + ord('0')) + s;
i := i div 10;
end;
if Negate then s + s;
if w < 127 then
    while w > ord(s[0]) do s + s;
AddStr(s);
end;

procedure PrintReal(w,d:longint);
var
    r : double;
    PackedReal : array[0..11] of byte;
    expnt,
    i : longint;
    s : string;
begin
    s := ""
    if UserProg then BlockUDRead(addr(r),a7,SizeOf(double))
    else move(SuperString,r.SizeOf(double));
    if (d<0) then d := 17;
    if d > 17 then d := 17;
d := -d;
{ no interrupts allowed}
inline($40e7/ {move.w sr,(a7)}
    $007c/$0700); {ori.w #$0700,st}
inline($2014/ {move.1 (a4),d0}
    $f22d/$5400/$0004/ {fmove.d 4(a5),fp0}
    $f22d/$7c00/$000c); {fmove.p fp0,$0c(a5),d0}
{ The packed string is now located in PackedReal}
{ no interrupts back on}
inline($46df); {move.w (a7)+,sr}
expt := (PackedReal[0] and $0f) * 100 + (PackedReal[1] div $10) * 10
    + (PackedReal[1] and $0f);
    if PackedReal[0] and $80 <> 0 then expnt := -expnt;
if w > 128 then w := 128;
d := -d;
if (w <= 0) or (expnt <= -15) or (expnt >= 15) then begin
    if PackedReal[0] and $80 <> 0 then s := ':' else s := ':' + ';
s := s + chr(PackedReal[3] mod $10 + ord('0')) + ':';
for i := 8 to 23 do
    if (i and 1) = 1 then
        s := s + chr(PackedReal[i div 2] and $0f + ord('0'))
    else s := s + chr(PackedReal[i div 2] div $10 + ord('0'));
end;
if PackedReal[0] and $40 <> 0 then s := s + '.' else s := s + '+';
for i := 1 to 3 do
  if (i and 1) = 1 then
    s := s + chr(PackedReal[i div 2] and $0f + ord('0'))
  else s := s + chr(PackedReal[i div 2] div $10 + ord('0'));
end else begin
  if PackedReal[0] and $80 <> 0 then s := '0';
  if expnt < 0 then begin
    s := s + '0.';
    i := 7;
    while (expnt < -1) and (d > 0) do begin
      s := s + '0'; inc(expnt); dec(d);
      end;
    while (d > 0) and (i <= 23) do begin
      if (i and 1) = 1 then
        s := s + chr(PackedReal[i div 2] and $0f + ord('0'))
      else s := s + chr(PackedReal[i div 2] div $10 + ord('0'));
      dec(d); inc(i);
      end;
  end else begin
    i := 7;
    while (expnt >= 0) and (i <= 23) do begin
      if (i and 1) = 1 then
        s := s + chr(PackedReal[i div 2] and $0f + ord('0'))
      else s := s + chr(PackedReal[i div 2] div $10 + ord('0'));
      dec(expnt); inc(i);
      end;
    while (expnt >= 0) and (i > 23) do begin
      s := s + '0';
      dec(expnt);
      end;
  end;
  if d <> 0 then s := s + '.';
  while (d > 0) and (i <= 23) do begin
    if (i and 1) = 1 then
      s := s + chr(PackedReal[i div 2] and $0f + ord('0'))
    else s := s + chr(PackedReal[i div 2] div $10 + ord('0'));
    dec(d); inc(i);
    end;
  while (d > 0) and (i > 23) do begin
    s := s + '0'; dec(d);
    end;
  end;
end;
while ord(s[0]) < w do s := ' ' + s;
AddStr(s);
end;

procedure PrintString(w:longint);
var s:string;
begin
  if UserProg then BlockUDRead(addr(s),a7,129)
else move(SuperString, s, 129);
if ord(s[0]) > 128 then s[0] := chr(128);
if w > 127 then w := 127;
while ord(s[0]) < w do s := ' ' + s;
AddStr(s);
end;

begi
UserProg := (StatusReg and $2000) = 0;
ThisPage := addr(ThisPage) and (-PageSize);
TimerStop;
case d0 of
  $00 : begin {program end}
    LeaveQ(ReadyQ);
    JoinQFlick(EtherQ, 0);
    GSendPacket^ [0] := $8e; {used to be $fe}
    LeaveQ(EtherQ);
    ProgramDone := true;
    Flick(0);
  end;
  $01 : {Run-Time error : d1 = error number}
    RunTimeError(3,'BPL Runtime Error #" + DecStr(d1) + ' at Address $' + HexStrL(ProgCntr, 8));
  $02 : ReturnFromProcess(a0);
    {End Process - aO = starting code address of process.
     Variable data starts at a0+6}
  $09 : FlushIOBuffer; {flush io}
  $0a, {reset file}
  $0b : begin {rewrite file}
    LeaveQ(ReadyQ);
    JoinQFlick(EtherQ, 0);
    {now, we are executing within an ethernet interrupt}
    if UserProg then begin
      if BReadUD(addr(ptr FileType(a0).Status)) <> 3
        then RunTimeError(3,'File Access Error at Address $'
          + HexStrL(ProgCntr, 8));
      Longint2Packet(LReadUD(addr(ptr FileType(a0).ElementSize)), 1);
      BlockUDRead(addr(GSendPacket ^ [5]),
        addr(ptr FileType(a0).Filename), 80);
    end else begin
      if TypedFile^ .Status <> 3 then
        RunTimeError(3,'File Access Error at Address $'
          + HexStrL(ProgCntr, 8));
      Longint2Packet(TypedFile^ .ElementSize, 1);
      move(TypedFile^ .FileName, GSendPacket ^ [5], 80);
    end;
    GSendPacket^ [0] := d0; { $0a or $0b}
    AwaitingEtherResponse := true;
    Flick(0);
    AwaitingEtherResponse := false;
if UserProg then begin
  LWriteUD(addr(ptr FileType(a0).Handle), Packet2Longint(1));
  BWriteUD(addr(ptr FileType(a0).Status), d0 - $0a + 1); {1 or 2}
end else begin
  TypedFile^.Handle := Packet2Longint(1);
  TypedFile^.Status := d0 - $0a + 1; {1 or 2}
end;
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;

$0c : begin {close file}
  if UserProg then begin
    if (BReadUD(addr(ptr FileType(a0).Status)) <> 1) and
        (BReadUD(addr(ptr FileType(a0).Status)) <> 2)
    then RunTimeError(3,'File Close Error at Address $'
                             + HexStrL(ProgCnt,8));
  end else begin
    if (TypedFile^.Status <> 1) and (TypedFile^.Status <> 2)
    then RunTimeError(3,'File Close Error at Address $'
                             + HexStrL(ProgCnt,8));
  end;
LeaveQ(ReadyQ);
JoinQFlick(EtherQ,0);
end;

$10 : begin {assign file}
  if UserProg then begin
    BWriteUD(addr(ptr FileType(a0).Status),3); {assigned file}
    LWriteUD(addr(ptr FileType(a0).ElementSize), d1);
    if BReadUD(a7) > 79 then BWriteUD(a7,79);
    BlockUDCopy(a7, addr(ptr FileType(a0).FileName), 80);
  end else begin
    TypedFile^.Status := 3; {assigned file}
    TypedFile^.ElementSize := d1;
    if ord(SuperString[0]) > 79 then SuperString[0] := chr(79);
    move(SuperString, TypedFile^.FileName, 80);
  end;
Relinquish;
end;

$13 : begin {clear input line}
  LeaveQ(ReadyQ);
  JoinQFlick(EtherQ,0);
StdPckt(1,$22);
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;

$14 : begin {print integer}
    ChkForFlush;
    PrintInt(d2,d1);
    Relinquish;
end;

$15 : begin {print real}
    ChkForFlush;
    PrintReal(d1,d2);
    Relinquish;
end;

$16 : begin {print char}
    ChkForFlush;
    PrintChar(d2,d1);
    Relinquish;
end;

$17 : begin {print string}
    ChkForFlush;
    PrintString(d1);
    Relinquish;
end;

$18 : begin {input integer}
    LeaveQ(ReadyQ);
    JoinQFlick(EtherQ,0);
    StdPckt(1,$22);
    AwaitingEtherResponse := true;
    Flick(0);
    AwaitingEtherResponse := false;
    d0 := Packet2Longint(1);
    LeaveQ(EtherQ);
    JoinQFlick(ReadyQ,0);
end;

$19 : begin {input char}
    LeaveQ(ReadyQ);
    JoinQFlick(EtherQ,0);
    StdPckt(1,$22);
    AwaitingEtherResponse := true;
    Flick(0);
    AwaitingEtherResponse := false;
    if UserProg then BWriteUD(a0,GRecvPacket^ [1])
    else alias3^[0] := chr(GRecvPacket^ [1]);
    LeaveQ(EtherQ);
    JoinQFlick(ReadyQ,0);
end;

$1a : begin {input string}
    LeaveQ(ReadyQ);
    JoinQFlick(EtherQ,0);
    StdPckt(1,$22);
AwaitingEtherResponse := true;
Flick(0);
AwaitingEtherResponse := false;
if GRecvPacket^ [1] > d1 then GRecvPacket^ [1] := d1;
if UserProg then BlockUDWrite(addr(GRecvPacket^ [1]),a0,
GRecvPacket^ [1]+1)
else move(GRecvPacket^ [1],alias3^ ,GRecvPacket^ [1]+1);
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;

$1b : begin {input real}
  LeaveQ(ReadyQ);
  JoinQFlick(EtherQ,0);
  StdPckt(1,$22);
  AwaitingEtherResponse := true;
  Flick(0);
  AwaitingEtherResponse := false;
  if UserProg then for i := 0 to 7 do
    BWriteUD(a7+7-i,GRecvPacket^ [i+1])
  else for i := 0 to 7 do alias4[7-i] := GRecvPacket^ [i+1];
  LeaveQ(EtherQ);
  JoinQFlick(ReadyQ,0);
end;

$1c : begin {Read typed file}
if UserProg
  then DataCount := LReadUD(addr(ptr FileType(a0).ElementSize))
else DataCount := TypedFile^.ElementSize;
LeaveQ(ReadyQ);
JoinQFlick(EtherQ,0);
while DataCount > 0 do begin
  GSendPacket^ [0] := dO ;
  if UserProg then
    Longint2Packet(LReadUD(addr(ptr FileType(a0).Handle)),1)
  else Longint2Packet(TypedFile^.Handle,1);
  SmallCount := 1450;
  if DataCount < 1450 then SmallCount := DataCount;
  Longint2Packet(SmallCount,5);
  AwaitingEtherResponse := true;
  Flick(0);
  AwaitingEtherResponse := false;
  if UserProg then BlockUDWrite(addr(GRecvPacket^ [1]),
    a1,SmallCount)
  else move(GRecvPacket^ [1],ptr byte (a1),SmallCount);
  inc(a1,SmallCount0);
  dec(DataCount,SmallCount);
  if DataCount > 0 then Flick(0);
end;
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;

$1d : begin {Write typed file}
DataCount := D1;

{***
  if UserProg
    then DataCount := LReadUD(addr(ptr FileType(aO).ElementSize))
  else
    DataCount := TypedFile^.ElementSize;
***}
LeaveQ(ReadyQ);
JoinQFlick(EtherQ,0);
while DataCount > 0 do begin
  GSendPacket^ [0] := dO;
  if UserProg then
    Longint2Packet(LReadUD(addr(ptr FileType(aO).Handle)), 1)
  else
    Longint2Packet(TypedFile^.Handle, 1);
  SmallCount := 1450;
  if DataCount < 1450 then SmallCount := DataCount;
  Longint2Packet(SmallCount, 5);
  if UserProg then
    BlockUDRead(addr(GSendPacket^ [9]), a1, SmallCount)
  else
    move(ptr byte (a1), GSendPacket^ [9], SmallCount);
  Longint2Packet(A1, 9 + SmallCount);
  if UserProg then
    Longint2Packet(PhysAddr(A1), 13 + SmallCount)
  else
    Longint2Packet(A1, 13 + SmallCount);
  inc(a1, SmallCount);
  dec(DataCount, SmallCount);
  if DataCount <> 0 then Flick(0);
end;
LeaveQ(EtherQ);
JoinQFlick(ReadyQ, 0);
end;

$20: {* Call resource procedure - Code address of the resource procedure is in aO. In the user space: The ID of the resource is on the stack, followed by the parameters}
CallResourceProc(aO,a7);

$21: {terminate resource procedure - no parameters}
ReturnFromResourceProc;

$30: if UserProg then begin {EOF} if (a1 <> -1) AND (BReadUD(addr(ptr FileType(a1).Status)) <> 1) then begin
do := 1;
Relinquish
end else begin
LeaveQ(ReadyQ);
JoinQFlick(EtherQ, 0);
GSendPacket^ [0] := $30;
if a1 = -1 then Longint2Packet(-1,1)
else Longint2Packet(LReadUD(addr(ptr FileType(a1).Handle)),1);
AwaitingEtherResponse := true;
Flick(0);
AwaitingEtherResponse := false;
do := Packet2Longint(1);
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;
end else begin
if (a1 <> -1) AND (TextFile^.Status <> 1)
then begin
do := 1;
Relinquish;
end else begin
LeaveQ(ReadyQ);
JoinQFlick(EtherQ,0);
GSendPacket^[0] := $30;
if a1 = -1 then Longint2Packet(-1,1)
else Longint2Packet(TextFile^.Handle,1);
AwaitingEtherResponse := true;
Flick(0);
AwaitingEtherResponse := false;
do := Packet2Longint(1);
LeaveQ(EtherQ);
JoinQFlick(ReadyQ,0);
end;
end;
$31 : begin {EOC - a0 points to global channel data structure

   return result in d0 (1=eoc, 0 = not eoc) }

do := ChkEOC(a0);
Relinquish;
end;
$32 : {Close channel - a0 points to global channel data structure,
   a1 points to data to send over channel prior to close,
   d1 contains size of the data type.}
WriteChan(a0,a1,d1,ProgCntr,true);
$33 : {reset channel - a0 points to global channel data structure }
ResetChan(a0,ProgCntr);
$34 : {rewrite channel - a0 points to global channel data structure }
RewriteChan(a0,ProgCntr);
$35 : {read channel - a0 points to global channel data structure,
   a1 points to data to read from channel,
   d1 contains size of the data type.}
ReadChan(a0,a1,d1,ProgCntr);
$36 : {write channel - a0 points to global channel data structure,
   a1 points to data to write to channel,
   d1 contains size of the data type.}
WriteChan(a0,a1,d1,ProgCntr,false);
$37 : {check channel status - a0 points to global channel data structure,
    Return d0 = 1 if the channel is ready to be read or written to.
Otherwise, return d0 = 0

begin
    d0 := CheckChannelStatus(a0);
    Relinquish;
end;

$38 : \text{Choose statement for select statement. In user data space, the acceptable addresses are on the stack. If the top longword is zero, the next address is the address of an else statement. It should only be executed if no other addresses are on the stack. After the last valid address, -1 is on the stack. This routine adjusts a7 to point to the -1. The address of the statement to execute should be stored at this address. If no statements are to be executed, set this return address to jump to the following address.}\)
    ChooseSelectStatement(a7);

$39 : \text{begin \{initialize channel - variable size in d1\}}
    d0 := InitChannel(d1);
    Relinquish;
end;

$3a : \text{begin \{deinitialize channel - channel id in d1\}}
    DelnitChannel(d1);
    Relinquish;
end;

$40 : \text{begin \{* Initialize resource variable - dl contains resource stack size, d2 contains resource variable size, return resource id number in d0\}}
    d0 := InitializeResource(d1,d2);
    Relinquish;
end;

$41 : \text{begin}
    DiscardResource(d1); \text{\{release resource variable\}}
    Relinquish; \text{\{d1 contains the process number\}}
end;

$51 : \text{begin \{get cpu/process information\}}
    \text{\{return cpu number in bits 0-7 of d0\}}
    \text{\{return process number in bits 8-31 of d0\}}
    d0 := MyCPUID + (addr(d0) and (-PageSize));
    Relinquish;
end;

$52 : \text{\{request read, request for write, release for template\}}
    \text{\{a0 points to template\}}
    \text{\{d1 = number of dimensions\}}
    \text{\{first word of structure is as follows:}\}
    0 : \text{release}
    1 : \text{read request}
    3 : \text{write request} \}
    \text{case TemplateFunc(a0,d1) of}
        1 : \text{RuntimeError(3,\text{Invalid Template Global Page Reference at '}}
            \text{\{Address $\}}
            \text{\{HexStrL(ProxCntr,8)\}}
        \text{end;
2 : RunTimeError(3,'Invalid Template Index Value at Address $'
+ HexStrL(ProgCntr,8));
3 : RunTimeError(3,'Too Many Shared Variable Requests'
+ 'at Address $'
+ HexStrL(ProgCntr,8));
end;
$53 : DoParallelCall(aO); {parallel statement start}
{a0 points to first process variable,
others are linked}
$55 : begin
  d0 := GetTemplateIdentifier;
  Relinquish;
end;
$60 : begin {Init shared variable }
  { Input Parameters : d1 = shared variable element size
    d2 = number of array dimensions
    These array index ranges are stored on the
    user stack pushed first index first, last
    index last (on top), lower limit pushed
    first, upper limit pushed last.
    Output Parameter : d0 = address of shared var control block }
  d0 := InitSharedVar(d1,d2,a7);
  Relinquish;
end;
$61 : begin {Discard shared variable}
  { Input Parameter : d0 = address of shared var control block }
  DiscardSharedVar(d0);
  Relinquish;
end;
$7f : begin
  d0 := GetRamPage;
  Relinquish;
end;
$ffffff0 : begin { Enable tracing }
  StatusReg := StatusReg OR $8000;
  Relinquish;
end;
$ffffff1 : begin { Disable tracing }
  StatusReg := StatusReg AND $7fff;
  Relinquish;
end;
$ffffff2 : begin
  DumpProcessPage(addr(d0),'Trap Dump Request');
  Relinquish;
end;
$ffffff3 : DO := PhysAddr(D1);
else RunTimeError(3,'Illegal TRAP #2 Instruction at Address $'
  + HexStrL(ProgCntr,8));
end;
end;
procedure Trace(FormatVector: word; ProgCntn: longint; StatusReg: word;
a7, a6, a5, a4, a3, a2, a1, a0,
d7, d6, d5, d4, d3, d2, d1, d0: longint); except;
begin
  Message('Instruction Trace, PC=' + HexStrL(ProgCntn, 8));
  Message('  D0=' + HexStrL(d0, 8)
    + ' D1=' + HexStrL(d1, 8)
    + ' D2=' + HexStrL(d2, 8)
    + ' D3=' + HexStrL(d3, 8));
  Message('  D4=' + HexStrL(d4, 8)
    + ' D5=' + HexStrL(d5, 8)
    + ' D6=' + HexStrL(d6, 8)
    + ' D7=' + HexStrL(d7, 8));
  Message('  A0=' + HexStrL(a0, 8)
    + ' A1=' + HexStrL(a1, 8)
    + ' A2=' + HexStrL(a2, 8)
    + ' A3=' + HexStrL(a3, 8));
  Message('  A4=' + HexStrL(a4, 8)
    + ' A5=' + HexStrL(a5, 8)
    + ' A6=' + HexStrL(a6, 8)
    + ' A7=' + HexStrL(a7, 8));
  FlushMessages;
end;

procedure StatTrace(FormatVector: word; ProgCntn: longint; StatusReg: word;
a7, a6, a5, a4, a3, a2, a1, a0,
d7, d6, d5, d4, d3, d2, d1, d0: longint); except;
begin
  Message('Statement Trace, PC=' + HexStrL(ProgCntn, 8));
  Message('  D0=' + HexStrL(d0, 8)
    + ' D1=' + HexStrL(d1, 8)
    + ' D2=' + HexStrL(d2, 8)
    + ' D3=' + HexStrL(d3, 8));
  Message('  D4=' + HexStrL(d4, 8)
    + ' D5=' + HexStrL(d5, 8)
    + ' D6=' + HexStrL(d6, 8)
    + ' D7=' + HexStrL(d7, 8));
  Message('  A0=' + HexStrL(a0, 8)
    + ' A1=' + HexStrL(a1, 8)
    + ' A2=' + HexStrL(a2, 8)
    + ' A3=' + HexStrL(a3, 8));
  Message('  A4=' + HexStrL(a4, 8)
    + ' A5=' + HexStrL(a5, 8)
    + ' A6=' + HexStrL(a6, 8)
    + ' A7=' + HexStrL(a7, 8));
  FlushMessages;
end;

procedure BusError(FormatVector: word; ProgCntn: longint; StatusReg: word;
a7, a6, a5, a4, a3, a2, a1, a0,
d7, d6, d5, d4, d3, d2, d1, d0: longint); except;
var
  DRegAlias : array[0..7] of longint absolute d0;
  ARegAlias : array[0..7] of longint absolute a0;
  S : string;
  i,Limit,
  A : longint;
begin
  RunTimeError(l,'Bus Error : PC = $' + HexStrL(ProgCntr, 8) + ' SP = $' + HexStrL(addr(StatusReg), 8) + ' SR = $' + HexStrL(StatusReg, 4) + ' Format $' + HexStrL(FormatVector div $1000, 1));
  A := addr(StatusReg);
  if (FormatVector AND $f000) = $a000 then Limit := $le
  else Limit := $5a;
  for i := 0 to Limit div 2 do begin
    S := 'SP + $' + HexStrL(2 * i, 2) + ' : ' + HexStrL(ptr word(A + 2 * i), 4); RunTimeError(0,S);
  end;
  S := ";
  for i := 0 to 3 do 
    S := S + ' D' + HexStrL(i,1) + ' = ' + HexStrL(DRegAlias[i],8) + ' ';
  RunTimeError(0,S);
  S := ";
  for i := 4 to 7 do 
    S := S + ' A' + HexStrL(i,1) + ' = ' + HexStrL(ARegAlias[i],8) + ' ';
  RunTimeError(0,S);
  if StatusReg AND $2000 = 0 then RunTimeError(2,'User Mode')
  else RunTimeError(2,'Supervisor Mode');
end;

procedure CPVError(FormatVector:word;ProgCntr:longint;StatusReg:word;
a7,a6,a5,a4,a3,a2,a1,a0,
d7,d6,d5,d4,d3,d2,d1,d0 : longint);exception;
var
  S : string;
  i,Limit,
  A : longint;
begin
  RunTimeError(1,Coprocessor Exception : PC = $' + HexStrL(ProgCntr, 8) + ' SR = $' + HexStrL(StatusReg, 4) + ' Format Vec = $' + HexStrL(FormatVector,4));
  A := addr(FormatVector) + 2;
Limit := 6;
for i := 0 to Limit-1 do begin
  if i AND $07 = 0 then S := HexStrL(8 + 2*i,2) + ': ';
  S := S + HexStrL(ptr word(A),4) + ' ';
  inc(A,2);
  if i AND $07 = 7 then begin
    RunTimeError(0,S);
    S := '';
  end;
end;
if S <> '' then RunTimeError(0,S);
if StatusReg AND $2000 = 0 then RunTimeError(2,'User Mode')
else RunTimeError(2,'Supervisor Mode');
end;

procedure ErrorVec(FormatVector:word;ProgCntr:longint;StatusReg:word);
  a7,a6,a5,a4,a3,a2,a1,a0;
  d7,d6,d5,d4,d3,d2,d1,d0 : longint);exception;
var
  i : longint;
  DRegAlias : array[0..7] of longint absolute dO;
  ARegAlias : array[0..7] of longint absolute aO;
  S : string;
  P : PProcessData;
begin
  RunTimeError(1,'Illegal Exception Error : PC = $' + HexStrL(ProgCntr,8)
    + ' SR = $' + HexStrL(StatusReg,4)
    + ' Format Vec = $' + HexStrL(FormatVector,4));
  S := '';
  for i := 0 to 3 do
    S := S + ' D' + HexStrL(i,1) + ' = ' + HexStrL(DRegAlias[i],8) + ' ';
  RunTimeError(0,S);
  S := '';
  for i := 4 to 7 do
    S := S + ' D' + HexStrL(i,1) + ' = ' + HexStrL(DRegAlias[i],8) + ' ';
  RunTimeError(0,S);
  S := '';
  for i := 0 to 3 do
    S := S + ' A' + HexStrL(i,1) + ' = ' + HexStrL(ARegAlias[i],8) + ' ';
  RunTimeError(0,S);
  S := '';
  for i := 4 to 7 do
    S := S + ' A' + HexStrL(i,1) + ' = ' + HexStrL(ARegAlias[i],8) + ' ';
  RunTimeError(0,S);
P := PDataPtr(addr(DO));
  RunTimeError(0,' Crashing at Process Page = $' + HexStrL(addr(DO) AND (-PageSize),8)
    + ' MMU Page = $' + HexStrL(P^ .MmuPages[UserDataLevelCIndex],8));
  if StatusReg AND $2000 = 0 then RunTimeError(2,'User Mode')
else RunTimeError(2,'Supervisor Mode');
end;
procedure AbortProcedure;exception;
begin
  ResetCPU;
end;

begin {main program}
  SetLED(LED1,MainLoopLEDSave);
  DisableMMU;
  if TestVer then begin
    inline($027c/$efff); { andi #$efff,SR }
    inline($2e7c/>InitialISR);
    inline($007c/$1700); { ori #$1700,SR }
    inline($2e7c/>InitialMSR);
  end else begin
    { First, clear each ram location. }
    inline(
      $207C/>RamBase/ { MOVEA.L #RamBase,A0 }
      $203C/>RamWords/ { MOVE.L #RamWords,D0 }
      { L: }
      $4298/ { CLR.L (A0)+ }
      $5380/ { SUBQ.L #1,D0 }
      $66FA); { BNE.S L }
  end;
  FlickDepth := 0;
  InitCPU;
  InitMem;
  InitMMU;
  InitQ(ReadyQ);
  InitQ(ParentProcQ);
  InitQ(ResourceWaitQ);
  InitQ(OpenChanQ);
  InitQ(SendChanQ);
  InitQ(RcvChanQ);
  InitQ(ShareQ);
  InitQ(EtherQ);
  UniqueTemplateID := 0;
  for i := 2 to $ff do
    SetExceptionVec(i,@ErrorVec);
  SetExceptionVec(31,@AbortProcedure);
  SetExceptionVec($22,@TRAP2);
  SetExceptionVec($20,@StatTrace);
  SetExceptionVec($09,@Trace);
  SetExceptionVec($02,@BusError);
  for i := 48 to 54 do
    SetExceptionVec(i,@CPVError);
  InitializeTimer;
  InitEthernet;
  InitCPUBoard;   { Board specific initialization. }
  ProgAddr := -1;
ptr longint(addr(OSMsgBlk)) := GetRamPage;
OSMsgBlk^.InPtr := 0;
OSMsgBlk^.OutPtr := 0;
ServeOSMsg := false;
DoTrace := false;
DoMsg := false;

MyWindow := 0;
if MyCPUID = 1 then InitGlobalRam;

InLine($007c/$0700/$027c/$f8ff); { enable interrupts }

SetLED(LED1,MainLoopLEDSave);
repeat until MyWindow <> 0;
SetLED(0,MainLoopLEDSave);

ProgramDone := false;
repeat
  CheckForNewProcesses;
  CheckForFinishedProcesses;
  CheckForReadyResources;
  CheckForReadyChannelOperations;
  if QReady(ReadyQ) and not RunTimeErrorPresent then begin
    SetLED(LED1,MainLoopLEDSave);
    ExecQ(ReadyQ);
    SetLED(0,MainLoopLEDSave);
  end;
until ProgramDone or RunTimeErrorPresent;

repeat until not RunTimeErrorPresent;

repeat until False;
end.
program EdgePreservingSmooth;

const
  ImageWidth = 199;
  ImageHeight = 199;
  MaxMaskSize = 9;
  NumMasks = 9;
  NumProcesses = 20;

  ResultRowsPerProcess = (ImageHeight - 3) div NumProcesses;
  SourceRowsPerProcess = ResultRowsPerProcess + 4;

type
  ImageRow = array[0..ImageWidth] of byte;

  ImageDesc = array[0..ImageHeight] of ImageRow;

  ResultOneProcess = array[1..ResultRowsPerProcess] of ImageRow;

  SourceOneProcess = array[1..SourceRowsPerProcess] of ImageRow;

  MaskDesc = record
    Size : longint;
    Elements : array[1..MaxMaskSize] of longint;
  end;

  MaskList = array[1..NumMasks] of MaskDesc;

  BoxDesc = array[1..25] of longint;

  Smoother = process(ImageIn : SourceOneProcess;
    ret ImageOut : ResultOneProcess;
    Masks : MaskList;
    In1, Out1 : longint);

procedure GetCpVForMask(var M : MaskDesc; var Box, SqrBox : BoxDesc;
var Cp, V : longint;

i : longint;

begin
  Cp := 0; V := 0;
  for i := 1 to M.Size do begin
    inc(Cp, Box[M.Elements[i]]);
    inc(V, SqrBox[M.Elements[i]]);
  end;
  dec(V, (Cp * Cp) div M.Size);
end;

var
  i, j, k, l : longint;
  Box,
  SqrBox : BoxDesc;
  BoxPtr,
  Cp, Cpt,
  V, Vt : longint;

begin
  for i := 1 to ResultRowsPerProcess do begin
    writeln('Doing Row #', i - 1 + Outl);
    ImageOut[i, 0] := 0;
    ImageOut[i, 1] := 0;
    ImageOut[i, ImageWidth - 1] := 0;
    ImageOut[i, ImageWidth] := 0;
  end;

  {*** Extract the 25 pixels of interest ***}
  for j := 2 to ImageWidth - 2 do begin
    BoxPtr := 1;
    for k := i to i + 4 do
      for l := j - 2 to j + 2 do begin
        Box[BoxPtr] := ImageIn[k, l];
        inc(BoxPtr);
      end;
  end;

  {*** Initialize SqrBox to contain squares of pixels of interest ***}
  for k := 1 to 25 do SqrBox[k] := Box[k] * Box[k];

  {*** Get mean, variance for each of the masks ***}
  GetCpVForMask(Masks[1], Box, SqrBox, Cp, V);
  for k := 2 to NumMasks - 1 do begin
    GetCpVForMask(Masks[k], Box, SqrBox, Cpt, Vt);
    if Vt < V then begin
      V := Vt;
      Cp := Cpt;
    end;
  end;
  GetCpVForMask(Masks[9], Box, SqrBox, Cpt, Vt);
{*** Output value corresponds to the minimum variance ***}
if (Vt div 9) < (V div 7) then
    ImageOut[i, j] := Cpt div 9
else
    ImageOut[i, j] := Cp div 7;
end;
end;
end;

var
Masks : MaskList;

{ ---------------------------------------------
   CREATING 5 X 5 WINDOW
 ---------------------------------------------}
procedure InitializeMasks;
var
    Mask : MaskDesc;
begin
    Mask.Size := 7;
    Mask.Elements[1] := 1;
    Masks[1] := Mask;

    Mask.Size := 7;
    Mask.Elements[1] := 2;
    Masks[2] := Mask;

    Mask.Size := 7;
    Mask.Elements[1] := 3;
    Masks[3] := Mask;

    Mask.Size := 7;
    Masks[4] := Mask;

    Mask.Size := 7;
end

Mask = [9]
MaskElements[0] = 12
MaskElements[1] = 16
MaskElements[2] = 14
MaskElements[5] = 6
MaskElements[6] = 7
Mask.Elements = [7]

Mask = [8]
MaskElements[0] = 17
MaskElements[1] = 16
MaskElements[3] = 12
MaskElements[5] = 7
MaskElements[6] = 6
MaskElements[7] = 9
Mask.Elements = [7]

Mask = [9]
MaskElements[0] = 22
MaskElements[1] = 16
MaskElements[3] = 12
MaskElements[5] = 7
Mask.Elements = [7]

Mask = [10]
MaskElements[0] = 24
MaskElements[1] = 19
MaskElements[2] = 18
MaskElements[3] = 17
MaskElements[5] = 19
Mask.Elements = [7]

Mask = [11]
MaskElements[0] = 25
MaskElements[1] = 24
MaskElements[2] = 20
MaskElements[3] = 19
MaskElements[4] = 18
MaskElements[5] = 14
Mask.Elements = [7]
var
  Imageln,
  ImageOut : ImageDesc;
F : file of ImageDesc;
Smoother : array[1..NumProcesses] of Smoother;
RemainderSmoother : Smoother;
i : longint;

begin
  InitializeMasks;
  assign(F, 'imgnsy');
  reset(F);
  read(F, Imageln);
  close(F);

  {*** Zero the first two and the last two rows of the result ***}
  FillChar(ImageOut[0], SizeOf(ImageRow), 0);
  FillChar(ImageOut[1], SizeOf(ImageRow), 0);
  FillChar(ImageOut[ImageHeight - 1], SizeOf(ImageRow), 0);
  FillChar(ImageOut[ImageHeight], SizeOf(ImageRow), 0);

  writeln('Elapsed Time : ');
  assign(F, 'smooth.img');
  rewrite(F);
  write(F, ImageOut);
  close(F);
end.
program ParSpec;

const
  BlockSize = 10000;
  BlockCount = 10;
  EntireSize = BlockCount * BlockSize;
  SampleSize = 2048;
  SampleCount = 200;
  RealBlockSize = 2 * SampleSize;
  FFTStride = 500;
  FFTCount = (EntireSize - SampleSize) div FFTStride + 1;
  NumProcesses = 15;

type
  SampleBlock = array[1..SampleSize] of integer;
  RealBlock = array[1..RealBlockSize] of real;
  EntireBlock = array[1..EntireSize] of integer;

DataReceiver = resource
  interface
    procedure Initialize;
    procedure StoreResults(DIn : SampleBlock; var StartPos : longint);
    procedure SaveResults;
  implementation
    var {*** Local resource variables ***}
      Samples : array[1..FFTCount] of SampleBlock;
      NextStartPos : longint;

    procedure Initialize;
    begin
      NextStartPos := NumProcesses;
    end;

    procedure StoreResults;
    begin
      Samples[StartPos + 1] := DIn;
      if NextStartPos >= FFTCount then StartPos := -1
      else begin
        StartPos := NextStartPos;
        inc(NextStartPos);
      end;
    end;

    procedure SaveResults;
    var
      F : file of SampleBlock;
i : longint;
begin
  assign(F, 'CumSpec.Dat');
  rewrite(F);
  for i := 1 to FFTCount do
    write(F, Samples[i]);
  close(F);
end;

FFTSolver = process(Data : EntireBlock;
  DR : DataReceiver;
  StartPos : longint);
var {*** Local process variables ***}
  RB : RealBlock;
  SB : SampleBlock;

PROCEDURE four1(VAR data: RealBlock; nn, isign: integer);
VAR
  ii, jj, n, mmax, m, j, istep, i: integer;
  wtemp, wr, wpr, wpi, wi, theta: double;
  temp, tempi: real;
BEGIN
  n := 2*nn;
  j := 1;
  FOR ii := 1 TO nn DO BEGIN
    i := 2*ii-1;
    IF (j > i) THEN BEGIN
      temp := data[j];
      tempi := data[j+1];
      data[j] := data[i];
      data[j+1] := data[i+1];
      data[i] := temp;
      data[i+1] := tempi
    END;
    m := n DIV 2;
    WHILE ((m >= 2) AND (j > m)) DO BEGIN
      j := j-m;
      m := m DIV 2
    END;
    j := j+m
  END;
  mmax := 2;
  WHILE (n > mmax) DO BEGIN
    istep := 2*mmax;
    theta := 6.28318530717959/(isign*mmax);
    wpr := -2.0*sqr(sin(0.5*theta));
    wpi := sin (theta);
    wr := 1.0;
    wi := 0.0;
    FOR ii := 1 TO (mmax DIV 2) DO BEGIN

\[ m := 2 \times i - 1; \]

\[
\text{FOR } jj := 0 \text{ TO } ((n-m) \text{ DIV } \text{ istep}) \text{ DO BEGIN}
\]
\[
i := m + jj \times \text{ istep};
\]
\[
j := i + \text{ mmax};
\]
\[
\text{tempr} := \text{ wr} \times \text{ data}[j] - \text{ wi} \times \text{ data}[j+1];
\]
\[
\text{tempi} := \text{ wr} \times \text{ data}[j+1] + \text{ wi} \times \text{ data}[j];
\]
\[
\text{data}[j] := \text{ data}[i] - \text{ tempr};
\]
\[
\text{data}[j+1] := \text{ data}[i+1] - \text{ tempi};
\]
\[
\text{data}[i] := \text{ data}[i] + \text{ tempr};
\]
\[
\text{data}[i+1] := \text{ data}[i+1] + \text{ tempi}
\]
\[
\text{END;}
\]
\[
\text{wtemp} := \text{ wr};
\]
\[
\text{wr} := \text{ wr} \times \text{ wpr} - \text{ wi} \times \text{ wpi} + \text{ wr};
\]
\[
\text{wi} := \text{ wi} \times \text{ wpr} + \text{ wtemp} \times \text{ wpi} + \text{ wi}
\]
\[
\text{END;}
\]
\[
\text{mmax} := \text{ istep}
\]
\[
\text{END}
\]

procedure CalcFFTfSP : longint);
var
  i : longint;
begin
{*** Create a complex input array ***}
for i := 1 to SampleSize do begin
  RB[2 * i - 1] := Data[i + SP * FFTStride];
  RB[2 * i] := 0.0;
end;

{*** Call the FFT routine ***}
Four1(RB, SampleSize, 1);

{*** Store the result back ***}
for i := 1 to SampleSize do
  SB[i] := sqrt(sqr(RB[2 * i - 1]) + sqr(RB[2 * i]));
end;

begin {*** Main body of process ***}
  while StartPos <> -1 do begin
    CalcFFT(StartPos);
    DR.StoreResults(SB, StartPos);
  end;
end;
end;

var
  Data : EntireBlock;
  i : longint;
  DR : DataReceiver;
  FFTS : array[1..NumProcesses] of FFTSolver;
procedure LoadData;
var
    F : file of EntireBlock;
begin
    assign(F, 'Data'); reset(F);
    read(F, Data);
    close(F);
end;

begin
    LoadData;
    writeln('^_^_^_');
    DR.Initialize;
    parallel
        for i := 1 to NumProcesses do
            FFTS[i](Data, DR, i - 1);
    end;
    writeln('Elapsed Time : & & & &');
    DR.SaveResults;
end.
Circuit Analysis Program

program ParCircuit;

const
  NumProcesses = 24;
  NumFreq = 120;
  InitialFreq = 20.0;
  FinalFreq = 20000.0;
  MaxNodes = 42;
  MaxMatrix = 2 * MaxNodes;
  MaxComponents = 100;
  Resistor = 1;
  Capacitor = 2;
  Inductor = 3;
  Gm = 4;
  VSource = 5;
  ISource = 6;

  {*** Operational amplifier characteristics ***}
  OpAmpInputZ = 100000000;
  OpAmpOutputZ = 100;
  OpAmpGain = 100000;

type
  {*** Data type describing a single component ***}
  ComponentDesc = record
    CompType : longint;
    Node1, Node2, Node3, Node4 : longint;
    Value : double;
  end;

  {*** Data type describing a circuit, a collection of components ***}
  CircuitDesc = record
    CompCount, Degree : longint;
    CompDesc : array[1..MaxComponents] of ComponentDesc;
  end;

  VectorDesc = array[1..MaxMatrix] of double;

  MatrixDesc = array[1..MaxMatrix] of VectorDesc;

  TransTableDesc = array[1..MaxNodes] of longint;
var
    Circuit : CircuitDesc;
    TransTable : TransTableDesc;
    InternalNode : longint;
    ProcessResults,
    ProcessFreqs : array[1..NumProcesses] of double;

procedure InitVariables;
begin
    Circuit.CompCount := 0;
    InternalNode := -1;
end;

procedure AddComponent(Typ,N1,N2, N3, N4 : longint; V : double);
var
    i : longint;
begin
    inc(Circuit.CompCount);
    i := Circuit.CompCount;
    Circuit.CompDesc[i].CompType := Typ;
    Circuit.CompDesc[i].Node1 := N1;
    Circuit.CompDesc[i].Node2 := N2;
    Circuit.CompDesc[i].Node3 := N3;
    Circuit.CompDesc[i].Node4 := N4;
    Circuit.CompDesc[i].Value := V;
end;

procedure AddOpAmp(NPlus,NMinus,NOut : longint);
begin
    AddComponent(Resistor,NPlus,NMinus,0,0,0,OpAmpInputZ);
    AddComponent(Resistor,NOut,0,0,0,OpAmpOutputZ);
    AddComponent(Gm,0,NOut,NPlus,NMinus,OpAmpGain / OpAmpOutputZ);
end;

procedure AddResistor(N1, N2 : longint; V : double);
begin
    AddComponent(Resistor,N1,N2,0,0,V);
end;

procedure AddCapacitor(N1, N2 : longint; V : double);
begin
    AddComponent(Capacitor,N1,N2,0,0,V);
end;

procedure AddVSource(N1, N2 : longint; V : double);
begin
    AddComponent(VSource,N1,N2,0,0,V);
end;

function TemporaryNode : longint;
begin
TemporaryNode := InternalNode;
dec(InternalNode);
end;

function RealNode(VirtualNode : longint) : longint;
var
  i : longint;
begin
  if VirtualNode = 0 then begin
    RealNode := 0; Exit;
  end;
  i := 1;
  while (i <= (Circuit.Degree div 2)) AND (TransTable[i] <> VirtualNode) do
  inc(i);
  if i > (Circuit.Degree div 2) then begin
    inc(Circuit.Degree,2);
    TransTable[Circuit.Degree div 2] := VirtualNode;
    RealNode := i;
  end;
end;

procedure TranslateVirtualNodes;
var
  i : longint;
begin
  Circuit.Degree := 0;
  for i := 1 to Circuit.CompCount do begin
    Circuit.CompDesc[i].Node1 := RealNode(Circuit.CompDesc[i].Node1);
    Circuit.CompDesc[i].Node2 := RealNode(Circuit.CompDesc[i].Node2);
    Circuit.CompDesc[i].Node3 := RealNode(Circuit.CompDesc[i].Node3);
    Circuit.CompDesc[i].Node4 := RealNode(Circuit.CompDesc[i].Node4);
  end;
end;

procedure SetGain(Gain, Pin : longint);
var
  G,
  M,
  N : longint;
begin
  if Gain > 0 then G := Gain
  else G := -Gain;
  case G of
    12 : M := $2f;
    11 : M := $2d;
    10 : M := $29;
    9  : M := $01;
    8  : M := $16;
    7  : M := $2a;
    6  : M := $12;
    5  : M := $02;
4 : \( M := $04; \)
3 : \( M := $08; \)
2 : \( M := $10; \)
1 : \( M := $20; \)
else \( M := 0; \)
end;
if \( Pin \leq 11 \) then begin
  if \( Gain < 0 \) then \( N := 4 \) else \( N := 1; \)
end else begin
  if \( Gain < 0 \) then \( N := 25 \) else \( N := 27; \)
end;
if \((M \text{ AND } 1) \neq 0\) then AddResistor(N,Pin,3e3);
if \((M \text{ AND } 2) \neq 0\) then AddResistor(N,Pin,8e3);
if \((M \text{ AND } 4) \neq 0\) then AddResistor(N,Pin,11e3);
if \((M \text{ AND } 8) \neq 0\) then AddResistor(N,Pin,16e3);
if \((M \text{ AND } 16) \neq 0\) then AddResistor(N,Pin,25e3);
if \((M \text{ AND } 32) \neq 0\) then AddResistor(N,Pin,55e3);
end;

procedure AddTunedCircuit(Pin : longint; Freq : double);
const
  \( Q_o = 3.5; \)
  \( Q_{12db} = 1.05; \)
var
  Ro,
  Cl,
  Rl,
  Co : double;
  N2,N3,N4 : longint;
begin
  N2 := TemporaryNode; N3 := TemporaryNode; N4 := TemporaryNode;
  Ro := 1590.0 / (Qo / Q12db - 1.0);
  Co := 1.0 / (2 * pi * Ro * Qo * Freq);
  Rl := 50e3;
  Cl := Ro*Qo*Co/Rl;
  AddCapacitor(Pin,N2,Co);
  AddCapacitor(N2,N3,Cl);
  AddResistor(N2,N4,Ro);
  AddResistor(N3,0,Rl);
  AddOpAmp(N3,N4,N4);
end;

type
  {*** Type definition for frequency supplying resource ***}
  FreqSourceDesc = resource
    interface
      procedure Initialize;
      procedure GetFrequency(ret W : double; ret Index : longint);
    implementation

var
FreqRatio,
CurrentW: double;
CurrentIndex: longint;

procedure Initialize;
begin
  CurrentW := InitialFreq;
  CurrentIndex := 1;
  FreqRatio := exp(ln(FinalFreq / InitialFreq) / (NumFreq - 1));
end;

procedure GetFrequency;
begin
  if CurrentIndex <= NumFreq then begin
    Index := CurrentIndex; W := CurrentW;
    inc(CurrentIndex);
    CurrentW := CurrentW * FreqRatio;
  end else begin
    Index := 0; W := 0.0;
  end;
end;

ResultDesc = array[1..NumFreq] of record
  Freq,
  Gain: double;
end;

(* Type definition for result storing resource *)
ResultStoreDesc = resource
interface
  procedure StoreGain(Index: longint; Freq, Gain: double);
  procedure RetrieveResult(ret Result: ResultDesc);
implementation
var
  Data: ResultDesc;

procedure StoreGain;
begin
  Data[Index].Freq := Freq;
  Data[Index].Gain := Gain;
end;

procedure RetrieveResult;
begin
  Result := Data;
end;
end;
type
CircuitProcDesc = process(Circuit: CircuitDesc;
   OutputNodeNum: longint;
   FSource: FreqSourceDesc;
   RStore: ResultStoreDesc);
var
   {*** Local process variables ***}
   Voltage,
   Current: VectorDesc;
   Matrix: MatrixDesc;
   W: double;
   Index: longint;

procedure BuildMatrix(W: double); {*** Nested procedure ***}
var
   i,j,k,l,m: longint;
   Val: double;
begin
   {*** Clear the Current vector. ***}
   for i := 1 to MaxMatrix do Current[i] := 0.0;
   {*** Clear the Matrix. ***}
   for i := 1 to MaxMatrix do begin
       Matrix[i] := Current;
       Matrix[i,i] := le-10;
   end;
   {*** Add resistors, capacitors, inductors, current sources. ***}
   for i := 1 to Circuit.CompCount do
     case Circuit.CompDesc[i].CompType of
       Resistor: begin
         if Circuit.CompDesc[i].Node1 <> 0 then begin
           j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
           Matrix[i+1,j+1] :=
               Matrix[i+1,j+1] - 1.0 / Circuit.CompDesc[i].Value;
         end;
         if Circuit.CompDesc[i].Node2 <> 0 then begin
           j := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
           Matrix[j+1,i+1] :=
               Matrix[j+1,i+1] - 1.0 / Circuit.CompDesc[i].Value;
         end;
         if (Circuit.CompDesc[i].Node1 <> 0) AND
            (Circuit.CompDesc[i].Node2 <> 0) then begin
           j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
           k := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
           Matrix[i,k] := Matrix[i,k] - 1.0 / Circuit.CompDesc[i].Value;
           Matrix[k,i] := Matrix[k,i] - 1.0 / Circuit.CompDesc[i].Value;
           Matrix[i+1,k+1] :=
               Matrix[i+1,k+1] + 1.0 / Circuit.CompDesc[i].Value;
           Matrix[k+1,i+1] :=
               Matrix[k+1,i+1] + 1.0 / Circuit.CompDesc[i].Value;
         end;
       end;
end;
Inductor,
Capacitor : begin
  if Circuit.CompDesc[i].CompType = Capacitor then
    Val := - Circuit.CompDesc[i].Value * W
  else
    Val := 1.0 / Circuit.CompDesc[i].Value / W;
  if Circuit.CompDesc[i].Node1 <> 0 then begin
    j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
    Matrix[j+1,j] := Matrix[j+1,j] + Val;
  end;
  if Circuit.CompDesc[i].Node2 <> 0 then begin
    j := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
    Matrix[j+1,j] := Matrix[j+1,j] + Val;
  end;
  if (Circuit.CompDesc[i].Node1 <> 0) AND
(Circuit.CompDesc[i].Node2 <> 0) then begin
    j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
    k := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
    Matrix[j,k+1] := Matrix[j,k+1] - Val;
    Matrix[j+1,k] := Matrix[j+1,k] - Val;
    Matrix[k,j+1] := Matrix[k,j+1] - Val;
    Matrix[k+1,j] := Matrix[k+1,j] - Val;
  end;
end;
ISource : begin
  if Circuit.CompDesc[i].Node1 <> 0 then begin
    j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
  end;
  if Circuit.CompDesc[i].Node2 <> 0 then begin
    j := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
  end;
end;
end;
{*** Add voltage controlled current sources. ***}
for i := 1 to Circuit.CompCount do
  if Circuit.CompDesc[i].CompType = Gm then begin
    if Circuit.CompDesc[i].Node1 <> 0 then begin
      j := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
      if Circuit.CompDesc[i].Node3 <> 0 then begin
        k := 2 * (Circuit.CompDesc[i].Node3 - 1) + 1;
        Matrix[j+1,k+1] := Matrix[j+1,k+1] + Circuit.CompDesc[i].Value;
      end;
    end;
    if Circuit.CompDesc[i].Node4 <> 0 then begin
      k := 2 * (Circuit.CompDesc[i].Node4 - 1) + 1;
    end;
  end;
Matrix[j+1,k+1] := Matrix[j+1,k+1] - Circuit.CompDesc[i].Value;
end;
end;
if Circuit.CompDesc[i].Node2 <> 0 then begin
  j := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
  if Circuit.CompDesc[i].Node3 <> 0 then begin
    k := 2 * (Circuit.CompDesc[i].Node3 - 1) + 1;
    Matrix[j+1,k+1] := Matrix[j+1,k+1] - Circuit.CompDesc[i].Value;
  end;
end;
if Circuit.CompDesc[i].Node4 <> 0 then begin
  k := 2 * (Circuit.CompDesc[i].Node4 - 1) + 1;
  Matrix[j+1,k+1] := Matrix[j+1,k+1] + Circuit.CompDesc[i].Value;
end;
end;
end;
end;

{*** Add voltage sources. ***}
for i := 1 to Circuit.CompCount do
if Circuit.CompDesc[i].CompType = VSource then begin
  j := 2 * (Circuit.CompDesc[i].Node2 - 1) + 1;
  for k := 1 to MaxMatrix do begin
    Matrix[j,k] := 0.0; Matrix[j+1,k] := 0.0;
  end;
  Matrix[j,j] := 1.0;
  Matrix[j+1,j+1] := 1.0;
  Current[j+1] := 0;
  if Circuit.CompDesc[i].Node1 <> 0 then begin
    k := 2 * (Circuit.CompDesc[i].Node1 - 1) + 1;
    Matrix[j,k] := -1.0;
    Matrix[j+1,k+1] := -1.0;
  end;
end;
end;

procedure ReduceRow(Row, TRow: longint); {*** Nested procedure ***}
var
  i: longint;
  Scale: double;
begin
  Scale := Matrix[Row,TRow] / Matrix[TRow,TRow];
  if Scale = 0.0 then Exit;
  Current[Row] := Current[Row] - Scale * Current[TRow];
  for i := TRow to Circuit.Degree do
    Matrix[Row,i] := Matrix[Row,i] - Scale * Matrix[TRow,i];
end;

procedure ReduceMatrix; {*** Nested procedure ***}
var
  i, j: longint;
begin
  for i := 1 to Circuit.Degree - 1 do
    for j := i + 1 to Circuit.Degree do ReduceRow(j,i);
end;

procedure SolveVoltage;  {*** Nested procedure ***}
var
  i, j : longint;
  ITemp : longint;
  XTemp : double;
begin
  {*** Triangularize the matrix. ***}
  ReduceMatrix;

  {*** Start at the bottom and solve up. ***}
  for i := Circuit.Degree downto 1 do begin
    Voltage[i] := Current[i];
    for j := i + 1 to Circuit.Degree do
    Voltage[i] := Voltage[i] / Matrix[i, i];
  end;
end;

function OutDB(Node : longint) : double;  {*** Nested function ***}
var
  VMag : double;
begin
  VMag := sqrt(sqr(Voltage[2*Node-1]) + sqr(Voltage[2*Node]));
  if VMag < 1e-12 then VMag := 1e-12;
  OutDB := ln(VMag) / ln(10.0) * 20.0;
end;

begin  {*** Main body of process ***}
  repeat
    {*** Get a frequency to solve circuit. ***}
    FSource.GetFrequency(W, Index);
    if Index <> 0 then begin
      {*** Solve the circuit for the frequency. ***}
      BuildMatrix(2 * pi * W);
      SolveVoltage;
      {*** Store the result. ***}
      RStore.StoreGain(Index, W, OutDB(OutputNodeNum));
    end;
    until Index = 0;
    {*** All frequencies are solved for - exit. ***}
  end;
end;

procedure ConstructCircuit;
begin
  InitVariables;
AddResistor(1,2,7.3e3);
AddResistor(3,4,7.3e3);
AddResistor(25,26,7.3e3);
AddResistor(26,27,7.3e3);

{*** Short between pins 2 & 3 ***}
AddResistor(2,3,0.001);

{*** Remove/Leave these four resistors for +6/12db ***}
AddResistor(1,0,3.4e3);
AddResistor(4,0,3.4e3);
AddResistor(25,0,3.4e3);
AddResistor(27,0,3.4e3);

{*** External components ***}
AddOpAmp(30,31,32);
AddOpAmp(33,34,35);
AddVSource(0,29,1);
AddResistor(29,30,27e3);
AddResistor(30,0,27e3);
AddResistor(2,0,10e3);
AddCapacitor(1,31,47e-6);
AddCapacitor(3,32,47e-6);
AddCapacitor(4,33,47e-6);
AddCapacitor(31,32,100e-12);
AddCapacitor(32,33,100e-12);
AddResistor(31,32,100e3);
AddResistor(32,33,100e3);
AddResistor(34,0,6.8e3);
AddResistor(34,35,6.8e3);

{*** EQ Components ***}
AddTunedCircuit(5,63);
AddTunedCircuit(6,160);
AddTunedCircuit(7,400);
AddTunedCircuit(8,1e3);
AddTunedCircuit(9,2.5e3);
AddTunedCircuit(10,6.3e3);
AddTunedCircuit(11,16e3);
SetGain(-10,5);
SetGain(0,6);
SetGain(0,7);
SetGain(12,8);
SetGain(0,9);
SetGain(0,10);
SetGain(0,11);
TranslateVirtualNodes;
end;
var
    ActualNode,
    i : longint;
    CircuitProcs : array[1..NumProcesses] of CircuitProcDesc;
    FSource : FreqSourceDesc;
    RStore : ResultStoreDesc;
    Result : ResultDesc;

begin
    {*** Initialize all variables describing the circuit. ***}
    ConstructCircuit;

    {*** Determine the node number of the output node. ***}
    ActualNode := RealNode(35);
    {*** Initialize the frequency source resource variable. ***}
    FSource.Initialize;
    write('\^\^\^\^');

    {*** Invoke the processes to solve the circuit. ***}
    parallel
        for i := 1 to NumProcesses do
            CircuitProcs[i](Circuit, ActualNode, FSource, RStore);
    end;

    {*** Get the results calculated by the processes. ***}
    RStore.RetrieveResult(Result);
    writeln('Elapsed Time : &\&\&');

    {*** Display the results. ***}
    for i := 1 to NumFreq do
        writeln(Result[i].Freq:8:2, ' : ', Result[i].Gain:7:3);
    end.
program ImageCompress;

const
  NumProcesses = 10;

  Quality = 15;
InputFileName = 'GS\LISAW.GS';
OutputFileName = 'GS\LISAW.JPG';
ImageW = 319;
ImageH = 199;

N = 8;
Nm1 = N - 1;
N2 = N * N;

NumStrips = (ImageH + 1) div N;

BlockSize = 1024;
PartialBlockSize = 8192;

type
  ImageRow = array[0..ImageW] of byte;

  ImageDesc = array[0..ImageH] of ImageRow;

  RealType = double;

  PixelStrip = array[0..Nm1] of ImageRow;

  PixelData = record
    StripNumber : longint;
    Pixels : PixelStrip;
  end;

  Matrix = array[0..Nm1, 0..Nm1] of RealType;

  QMatrix = array[0..Nm1, 0..Nm1] of longint;

  OutputBlock = array[1..BlockSize] of byte;

  PartialOutput = record
    Count,
    StripNumber : longint;
    ExtraVal : longint;
    Data : array[1..PartialBlockSize] of byte;
  end;

  PartialGroup = array[1..NumStrips] of PartialOutput;
Position = record
X,
Y : longint;
end;

ScrambleDesc = array[1..N2] of Position;

PixelChan = channel of PixelData;

PixelChans = array[1..NumProcesses] of PixelChan;

PartialChan = channel of PartialOutput;

PartialChans = array[1..NumProcesses] of PartialChan;

JPGProcDesc = process(CIn : PixelChan; COut : PartialChan; Num : byte);

var {*** Local process variables ***}
C, { Cosine matrix }
Ct : Matrix; { Transposed cosine matrix }
PM, { Pixel matrix }
DCT : Matrix; { Transformed matrix }
QM : QMatrix; { Quantization matrix }
SD : ScrambleDesc; { Zig-zag description }
PD : PixelData; { Input for one pixel strip }
PO : PartialOutput; { Output for one pixel strip }
ZeroCount : longint; { Run-length encode count }

procedure Initialize;
{*** Initialize all local variables, open channels ***}
var 
i, j, k : longint;
Down : boolean;
begin 
{ Initialize the cosine matrices }
for j := 0 to N - 1 do begin
  C[0, j] := 1.0 / sqrt(N);
  Ct[j, 0] := C[0, j];
end;

for i := 1 to N - 1 do
  for j := 0 to N - 1 do begin
    C[i, j] := sqrt(2.0 / N) * cos((2 * j + 1) * i * pi / (2.0 * N));
    Ct[j, i] := C[i, j]
  end;

{ Initialize the quantization matrix }
for i := 0 to N - 1 do 
  for j := 0 to N - 1 do 
    QM[i, j] := 1 + (1 + i + j) * Quality;

{ Initialize the zig-zag description }
i := 0; j := 0; Down := True;
for k := 1 to N2 do begin
  SD[k].Y := i; SD[k].X := j;
  if Down then begin
    if i = N - 1 then begin
      inc(j); Down := False;
    end else begin
      if j = 0 then begin
        inc(i); Down := False;
      end else begin
        inc(i); dec(j);
      end;
    end;
  end else begin
    if j = N - 1 then begin
      inc(i); Down := True;
    end else begin
      if i = 0 then begin
        inc(j); Down := True;
      end else begin
        dec(i); inc(j);
      end;
    end;
  end;
end;

{ Open the channels }
Reset(CIn);
Rewrite(COut);
end;

procedure ExtractMatrixfCol : longint;
var PS : PixelStrip;
var M : Matrix;
{*** Extract one 8x8 pixel matrix ***}
var
i, j : longint;
begin
  for i := 0 to N - 1 do
    for j := 0 to N - 1 do
      M[i, j] := 1.0 * PS[i, Col + j] - 128;
end;

procedure MatrixMult(var M1, M2, MOut : Matrix);
{*** Multiply M1 * M2, store result in MOut ***}
var
i, j, k : byte;
T : RealType;
begin
  for i := 0 to N - 1 do
    for j := 0 to N - 1 do begin
T := 0.0;
for k := 0 to N - 1 do
  T := T + M1[i, k] * M2[k, j];
  MOut[i, j] := T;
end;
end;

procedure CalcDCT(var MIn, MOut: Matrix);
{*** Calculate the DCT of the pixels in MIn, store result in MOut ***}
var
  Temp : Matrix;
begin
  MatrixMult(MIn, Ct, Temp);
  MatrixMult(C, Temp, MOut);
end;

procedure Quantize(var M: Matrix);
{*** Quantize the result of the DCT ***}
var
  i, j : longint;
begin
  for i := 0 to N - 1 do
    for j := 0 to N - 1 do
      M[i, j] := M[i, j] / QM[i, j];
end;

procedure NibbleOut(N: longint);
begin
  inc(PO.Count);
  if PO.Count AND 1 <> 0 then
    PO.Data[1 + PO.Count div 2] := N
  else
    PO.Data[PO.Count div 2] := ($0f AND PO.Data[PO.Count div 2])
    OR ((N * $10) AND $f0);
end;

procedure EmitInteger(V: longint);
{*** Split V into nibbles, send to output ***}
begin
  if V > 0 then
    while V > 3 do begin
      NibbleOut(V AND $07);
      V := (V AND $ffffff8) div 8;
    end
  else
    while V < -4 do begin
      NibbleOut(V AND $07);
      V := (V AND $ffffff8) div 8;
    end;
  NibbleOut((V AND $07) OR $08);
end;
procedure ValOut(V: longint);
{*** Append value V to output data block ***}
begin
  if V = 0 then begin
    inc(ZeroCount);
    Exit;
  end;
  if ZeroCount <> 0 then begin
    EmitInteger(0);
    EmitInteger(ZeroCount);
    ZeroCount := 0;
  end;
  EmitInteger(V);
end;

procedure OutputResults(var M: Matrix);
{*** Round the values in M, store in zig-zag order ***}
var
  i, j: longint;
begin
  for i := 1 to N div 2 do
    ValOut(Round(M[SD[i].Y, SD[i].X]));
end;

procedure EndOfStrip;
{*** Pad the end of the output data block ***}
begin
  if ZeroCount <> 0 then begin
    EmitInteger(0);
    EmitInteger(ZeroCount);
  end;
  if PO.Count AND 1 = 0 then Exit;
  NibbleOut($08);
  NibbleOut($00);
  NibbleOut($08);
end;

procedure CompressPixelStrip(var PD: PixelData; var PO: PartialOutput);
{*** Produce compressed data for one pixel strip ***}
var
  PM, { Pixel matrix }
  DCT : Matrix; { Transformed matrix }
  i: longint;
begin
  PO.StripNumber := PD.StripNumber;
  PO.Count := 0;
  ZeroCount := 0;
  if PD.StripNumber = 1 then
    ValOut(Quality);
  for i:= 0 to (ImageW + 1) div N - 1 do begin
    ExtractMatrix(i * N, PD.Pixels, PM);
CalcDCT(PM, DCT);
Quantize(DCT);
OutputResults(DCT);
end;
EndOfStrip;
end;

begin {*** Main body of process ***}
Initialize;
repeat
  read(CIn, PD);
  if PD.StripNumber <> -1 then begin
    CompressPixelStrip(PD, PO);
    write(COut, PO);
  end;
  until PD.StripNumber = -1;
end;
end; {*** End of process definition ***}

InputProcess = process(Image : ImageDesc; PChans : PixelChans);
var
  CurrentStrip,
  DoneCount,
  i : longint;
  PD : PixelData;

procedure HandleSendProc(Num : longint);
begin
  if CurrentStrip > NumStrips then begin
    PD.StripNumber := -1;
    inc(DoneCount);
  end else begin
    PD.StripNumber := CurrentStrip;
    for i := 0 to N - 1 do
      PD.Pixels[i] := Image[N * (CurrentStrip - 1) + i];
    inc(CurrentStrip);
    write(PChans[Num], PD);
  end;
end;

begin {*** Main body of process ***}
\{ Open the channels \}
for i := 1 to NumProcesses do begin
  Rewrite(PChans[i]);
end;
CurrentStrip := 1;
DoneCount := 0;
while DoneCount < NumProcesses do
  \{ Find a ready process, send it some pixels \}
  select
    PChans[1] : HandleSendProc(1);
PChans[2] : HandleSendProc(2);
PChans[3] : HandleSendProc(3);
PChans[4] : HandleSendProc(4);
PChans[5] : HandleSendProc(5);
PChans[6] : HandleSendProc(6);
PChans[7] : HandleSendProc(7);
PChans[8] : HandleSendProc(8);
PChans[9] : HandleSendProc(9);
PChans[10] : HandleSendProc(10);
end;
end;  {*** End of process definition ***}

OutputProcess = process(PChans : PartialChans);
var
  i,
  PartialCount : longint;
  Partials : PartialGroup;
  OnePartial : PartialOutput;
  OutputBlocks : array[1..32] of OutputBlock;

procedure StoreCompressedData;
var
  i,
  TotalSize,
  NumBlocks,
  P : longint;
  F : file of OutputBlock;
begin
  {*** Combine the data, one partial at a time. ***}
  TotalSize := 0;
  for i := 1 to NumStrips do begin
    Move(Partials[i].Data,
         ptr byte (TotalSize + addr(OutputBlocks)),
         Partials[i].Count div 2);
    inc(TotalSize, Partials[i].Count div 2);
  end;

  NumBlocks := (TotalSize + BlockSize - 1) div BlockSize;

  {*** Zero pad the end of the data ***}
  if TotalSize mod BlockSize <> 0 then
    for i := TotalSize mod BlockSize + 1 to BlockSize do
      OutputBlocks[NumBlocks][i] := 0;

  {*** Display the execution time. ***}
  writeln('Elapsed Time : & & & & ');

  {*** Open the file, write out the results ***}
  assign(F, OutputFileName);
  rewrite(F);
for i := 1 to NumBlocks do
  write(F, OutputBlocks[i]);
  close(F);
end;

procedure HandleRcvProc(N : longint);
begin
  inc(PartialCount);
  read(PChans[N], OnePartial);
  Partials[OnePartial.StripNumber] := OnePartial;
end;

begin   {*** Main body of process ***}
  for i := 1 to NumProcesses do
    reset(PChans[i]);
    PartialCount := 0;
    while PartialCount < NumStrips do
      { Find a ready process, receive its partial result }
      select
        PChans[1] : HandleRcvProc(1);
        PChans[2] : HandleRcvProc(2);
        PChans[3] : HandleRcvProc(3);
        PChans[4] : HandleRcvProc(4);
        PChans[5] : HandleRcvProc(5);
        PChans[6] : HandleRcvProc(6);
        PChans[7] : HandleRcvProc(7);
        PChans[8] : HandleRcvProc(8);
        PChans[9] : HandleRcvProc(9);
        PChans[10] : HandleRcvProc(10);
      end;
      StoreCompressedData;
    end;   {*** End of process definition ***}
end;

var
  InProc : InputProcess;
  InChans : PixelChans;
  OutProc : OutputProcess;
  OutChans : PartialChans;
  Image : ImageDesc;
  i : longint;

procedure ReadImage;
  {*** Read in the entire image. ***}
var
  F : file of ImageDesc;
begin
  assign(F, InputFileName);
  reset(F);
  read(F, Image);
close(F);
end;

begin
  ReadImage;
  write('^

parallel
  for i := 1 to NumProcesses do
    J Peggers[i](InChans[i], OutChans[i], i);
    InProc(Image, InChans);
    OutProc(OutChans);
  end;
end.