RP-ECU: Development of a rapid-prototyping system for diesel engine controls

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Aaron Bonnell-Kangas, B.S.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2016

Dissertation Committee:

Giorgio Rizzoni, Advisor

Marcello Canova

Steven Bibyk
Abstract

An open rapid-prototyping engine control system is developed based on a commercial platform and implemented on a 2L four-stroke diesel engine at the Ohio State University Center for Automotive Research. The procedure for setting up basic diesel engine controls on an unknown engine is summarized, and a generalized software architecture for portable controls modeling is outlined. An outline is provided of the documentation generated in the course of the project.
Acknowledgements

Thanks to Prof. Canova for his unyielding support and patience, even in the many times when I stumbled.

Thanks to the other professors and staff at CAR who support me and answer my incessant questions. Prof. Midlam-Mohler deserves special mention for his mentorship, not to mention all of the equipment that I liberated from the EcoCAR team. Don Williams experience was crucial to the entire mechanical design and fabrication effort.

General Motors hardware donations and technical support made the project possible thanks especially to Y.Y., Arnie, and the others at GM RD who took the time to answer questions and give advice.

Huge thanks to the other students working on the project – Sean Harrington, Arnez Franklin, Kang Rong, Junqiang Zhou, Zhaoxuan Zhu – for their diligent work on the broad range of tasks they were required to take on in the course of an engine setup project. Zhaoxuan must be mentioned specially because of the extra work he has done to deal with some of the most difficult problems in the project.

Buckeye Current provided hand tools and spare parts throughout the duration of the project, defined my experience at Ohio State, and gave me a home for five years.

Finally, I feel extremely grateful to have terrific friends, family, and girlfriend in my life who support me and challenge me through all of my trials.
Vita

May 2009 ........................................ Columbus Alternative High School

2011 - 2016 ......................... Buckeye Current Electric Motorcycle Race Team

2014 ............... B.S. Electrical and Computer Engineering, Ohio State University

2014 to present  Graduate Research Associate, Center for Automotive Research, The Ohio State University

Fields of Study

Major Field: Electrical and Computer Engineering
# Table of Contents

Abstract ii

Acknowledgements iii

Vita iv

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 What are engine controls? 1

1.2 Engine controls at OSU CAR 2

1.3 This document 2

2 Background 3

2.1 Prototyping and developing engine controls 3

2.1.1 Modern controls development workflow 3

2.1.2 Rapid controls prototyping (RCP) 4

2.2 This project 6

2.2.1 Scope and objectives 6

2.2.2 History and timeline of development 6
3 Development Process

3.1 Overview ............................................................... 8
3.2 Engine platform selection ........................................... 8
3.3 Rapid controls prototyping and data acquisition platform selection ...... 9
3.4 Definition of control requirements .................................. 10
  3.4.1 Basic diesel engine control structure ......................... 11
  3.4.2 Safety checks and diagnostics .................................. 11
3.5 Controls development .................................................. 12
  3.5.1 Required components ............................................ 12
  3.5.2 Sensor development .............................................. 12
  3.5.3 Actuator development ........................................... 13
  3.5.4 Controller development ......................................... 14
3.6 Integration and validation ............................................. 15
  3.6.1 Moving controller models from simulation to ECU platform ...... 15
  3.6.2 System-level validation ......................................... 17

4 Software Architecture ................................................. 19

4.1 Problem overview: software architecture goals and requirements ........ 19
  4.1.1 Functional requirements ...................................... 19
  4.1.2 Goals .............................................................. 20
4.2 Solution overview: software architecture ................................ 20
  4.2.1 Types of information .......................................... 21
  4.2.2 ASW software structure ....................................... 22
  4.2.3 Controller specifications ...................................... 24
  4.2.4 Sensor and actuator interface specifications .................... 27
4.2.5 ASW interfaces ................................................. 27

5 Workflow and toolchain .................................. 30

5.1 Toolchain ....................................................... 30

5.1.1 Core tools .................................................... 31

5.1.2 Data flow ..................................................... 31

5.2 Workflow ......................................................... 32

5.2.1 Development of ECU controls for a new engine. .... 32

5.2.2 Collaboration using distributed version control system (DVCS) ... 41

5.3 Documentation ................................................ 44

6 Conclusions ..................................................... 47
List of Tables

Table 5.1: Input and sensor specifications for top-level diagram shown in figure 5.4. .............................. 37

Table 5.2: Output and actuator specifications for top-level diagram shown in figure 5.4. ............................. 38

Table 5.3: Common Git terminology for software developers. ......................................................... 43
List of Figures

Figure 3.1: A20DTH engine airpath schematic, with measurement points, sensors, and actuators indicated. ................................. 9

Figure 3.2: Comparison of ECU platforms evaluated for RP-ECU project, with pros and cons. ..................................................... 10

Figure 4.1: The four classes of software information that make up the ECU software, and the origin of each component. Application software (ASW) code and data is generated from a Simulink model of the controller. Base software (BSW) information is supplied by the ECU platform vendor, and modifications to the BSW data are stored early in the development cycle. .................................................. 22

Figure 4.2: The hierarchy of the ASW model. A single container model contains one or more control loops, each of which operates independently and asynchronously. Each control loop is broken down into sensors, controllers, and actuators. .................................................. 23

Figure 4.3: Controller models in context of the ASW software hierarchy. Controllers are composed of a Simulink model, an initialization script, and one or more data files. .................................................. 24

Figure 4.4: Example of a single controller block with physical signal inputs and outputs. ......................................................... 25
Figure 4.5: Controllers expose measurements and calibrations to the developer, which are accessible using the calibration tool. Measurements are read-only, and allow observation of variables inside the ASW. Calibrations allow data inside the ASW to be altered at runtime.

Figure 5.1: Flow of information between the various software tools during the software compilation and ECU flashing processes.

Figure 5.2: Diagram of development workflow, with deliverables listed at each step.

Figure 5.3: Example of a single controller with inputs and outputs. The inputs and outputs are defined in terms of their physical units.

Figure 5.4: Example of a top-level control structure with inputs, outputs, and controllers.

Figure 5.5: Data flow inside a sensor block. The physical value lookup is shown here as a 1-D lookup table, but it might also be a mathematical equation or more complex mapping.
Chapter 1: Introduction

1.1 What are engine controls?

Since the passage of the Clean Air Act in 1970, and the Energy Policy and Conservation Act in 1975, automotive design has been influenced significantly by legal restrictions on tailpipe emissions and fuel consumption. Limitations on the emission of harmful gases, such as nitrous oxides and carbon monoxide, as well as the economic and regulatory incentivization of fuel economy, have driven the adoption of technologies that allow for more precise control over the combustion process and treatment of its byproducts.

Since then, and with the widespread adoption of technologies such as high-pressure fuel injection, exhaust-gas recirculation (EGR), variable valve timing (VVT), and electronically controlled ignition, the number of systems involved in the modern combustion engine has increased sharply in the past 50 years. This increase in complexity has been enabled by the use of digital electronic engine management systems which use feedback control algorithms to monitor and regulate engine operation.

As the complexity of this engine management software has increased, market pressures have driven interest in reducing the cost and time required to develop new controls algorithms and validate their behavior in the wide range of operating conditions they are expected to perform in. In the beginning of the 21st century, model-based design became the favored method for the design, prototyping, and validation of control systems. Rapid controls prototyping (RPC) tools are a key component of the model-based design workflow.
1.2 Engine controls at OSU CAR

The Center for Automotive Research (CAR), at Ohio State University, has been deeply involved in the development of engine controls for over 15 years. Historically, prototype controls have been developed using bypass rapid controls prototyping methods, requiring considerable non-recurring engineering investment for each separate project undertaken.

In 2014, the CAR Industrial Consortium (CAR IC) commissioned the CAR Rapid-Prototyping ECU (RP-ECU) project. The goal of the project was to create the necessary hardware and software capabilities and engineering processes to develop control systems that can operate a wide variety of engine and powertrain platforms with minimal setup and recalibration time. The project was projected to streamline research operations and reduce costs by providing a consolidated platform for engine setup and re-use of development work.

A new engine controls project at OSU CAR, based on an available turbodiesel engine, was selected as the first trial of the newly developed system.

1.3 This document

This document provides an overview of the controls development process for the first engine to be fired using the RP-ECU system, as well as a description of the procedures and documentation that were generated during the development of the system.

Additional documentation was created in order to facilitate development work for future users and streamline the development process. This document provides an index of the documentation that was created and a summary of its purpose.
Chapter 2: Background

2.1 Prototyping and developing engine controls

Historically, many organizations have used a document-based or specification-based design workflow to implement complex controls. In the model-based design workflow, all engineering tasks (design, prototyping, validation) are based around an executable system model.

2.1.1 Modern controls development workflow

The model-based testing workflow is representative of many of the recent developments in the field. Using model-based testing, controllers are developed and tested in different ways during each of several development stages:

- Model-in-the-loop (MIL): interpreted simulation models of the plant and the controller algorithm are tested against one another on a computer.

- Software-in-the-loop (SIL): an interpreted simulation model of the plant is tested with a compiled version of the controller algorithm. Both the plant model and the controller model are running on a relatively powerful host computer.

- Processor-in-the-loop (PIL): an interpreted simulation model of the plant is tested against a compiled version of the controller that is running on an actual microprocessor. PIL simulation injects and extracts data from the processor digitally after the sensor layer, so it is not necessary to generate physical signals for each sensor.
Hardware-in-the-loop (HIL): a simulation model of the plant is executed on specialized simulator hardware, which runs the model in real time and generates

2.1.2 Rapid controls prototyping (RCP)

Rapid controls prototyping (RCP) is a process, and the associated tools, for quickly creating functional prototypes of engine controllers. It is a key component of the "X-in-the-loop" (xIL) and model-based design (MBD) workflows.

The key task of RCP in the context of the MBD workflow is the generation of a functional controller from a simulation model of that controller’s behavior. When this is effectively implemented, it reduces the costs and time associated with controls development:

• Compared with a traditional document-based prototyping process, RCP reduces the development effort and communication required between a theoretical design and a functional prototype. This in turn reduces the risk of errors introduced by communication and translation during the development process.

• RCP allows for a reduction in prototyping cost, since it automates the prototyping process, and allows for control models to be tested earlier and faster.

Two categorizations are useful for describing the nature of RCP tools: bypass RCP vs. open RCP, and traditional vs. on-target RCP.

Bypass vs. ”open” controls

From the perspective of the controls developer, RCP tools can be divided into ”bypass” tools and ”open” tools.

Bypass RCP tools involve modifying an existing controller to change its behavior. For example, a bypass RCP tool might modify an existing engine controller by changing the
behavior of its exhaust-gas recirculation (EGR) control loop. Typically, this requires pre-
arranged access to the control software, in order to make modifications, or the ability to 
selectively disable the subcomponents of the existing controller. Although the developer 
has specific, pre-arranged access to these features, she typically does not have the ability to 
edit the software internals at will.

In contrast, open RCP tools allow the developer full access to the software structure and 
behavior, which can be modified as necessary. Rather than selectively disabling components 
of the software and re-recreating them with the necessary changes, the controls developer 
can make the modifications directly to the software in place.

Access to proprietary algorithms and software is a major obstacle to open RCP: having 
access to the source code or other controls specification for editing requires that the devel-
oper has access to the proprietary algorithms that it has implemented. For that reason, 
bypass RCP is often used to develop modifications to existing controls provided by a third 
party.

**Traditional vs. on-target RCP**

Implementation of the prototype can be achieved on two different types of hardware.

Traditional RCP tools feature high-powered computational hardware to execute the 
simulation model in real-time when compared to the production target hardware. These 
systems have greater computational ability and more memory than the engine control unit 
(ECU) that will ship with a production vehicle. Often, they also provide high-speed data 
buses for attaching additional input and output (I/O) hardware, providing flexibility and 
expandability.

Alternatively, the prototype can be developed to run on hardware that is very similar 
to the final production hardware. Compared to traditional RCP hardware systems, these
"on-target" RCP tools have more constraints on processing power, memory, and I/O.

Although traditional RCP tools are more powerful and flexible than on-target tools, they typically have a higher cost reflecting their sophistication and limited market. On-target tools based on production hardware can leverage existing economies of scale. Additionally, if a goal of the prototyping process is to validate the controller’s behavior in a realistic resource-constrained environment, traditional RCP tools do not accurately reflect the behavior of the final vehicle’s ECU.

2.2 This project

2.2.1 Scope and objectives

The CAR RP-ECU project was developed at Ohio State’s Center for Automotive Research for the CAR Industrial Consortium (CAR IC). The intent of the project was to benefit Consortium members by providing enhanced controls development capabilities to the Center. More specifically, the project’s goal is to "build a framework that provides the ability to prototype, implement and experimentally verify control algorithms for engine air path, fuel path, and other systems".

After the project’s charter in spring 2015, the initial goal was quickly set as the implementation of a basic diesel engine controls framework on a GM A20DTH engine.

2.2.2 History and timeline of development

- January 2015 - Project charter
- March 2015 - Control system selection
- Summer 2015 - Initial development efforts with gasoline FlexECU, wiring harness specification; design and assembly of electrical system
• July-November 2015 - Mechanical setup of engine in dynamometer test cell; design and installation of airpath and cooling system

• September 2015 - Diesel FlexECU received

• November 2015 - March 2016 Development and testing of sensor and actuator interfaces

• Jan 2016 - Feb 2016 - Development and simulation of airpath and fuel controllers

• April 2016 - Integration testing of fuel rail pressure controller

• May 2016 (projected) - First engine firing
Chapter 3: Development Process

3.1 Overview

A number of tasks were outlined in order to configure the RP-ECU system to fire the first testbed engine. This section provides a high-level overview of the procedure for selecting an engine, mapping out the required tasks, and finally developing a set of basic diesel engine controls to demonstrate the system’s capabilities.

3.2 Engine platform selection

After the charter of the project, the first engine platform was quickly selected as a small turbodiesel engine, the GM A20DTH. The engine was made available by a donation from General Motors.

The A20DTH is a four-cylinder, four-stroke diesel engine with 1956 cm$^3$ displacement, used in passenger sedans for the American and European markets. It features a variable-geometry turbine (VGT) turbocharger, high-pressure exhaust gas recirculation (EGR), common-rail direct fuel injection, and a throttle actuator.

Figure 3.1 shows the engine airpath with key sensors and actuators labeled.
3.3 Rapid controls prototyping and data acquisition platform selection

We evaluated commercially available RCP systems from three major vendors: DSpace MicroAutoBox, National Instruments Drivven, and ETAS FlexECU.
National Instruments Driven engine control system
Full-authority engine controller based on a PXI platform. Includes open-source engine control code based on LabVIEW that serves as a steady-state engine controller and a template for more advanced applications.

+ Turn-key solution for engine testing and research.
+ Supplied with built-in libraries for diesel engine air path control and fuel injection control.
– Control software is LabVIEW based (limited interfacing with MATLAB/Simulink)
– Mostly targeted at steady-state engine combustion and performance analysis.

dSPACE MicroAutoBox + RapidPRO
Modular rapid-prototyping hardware (MicroAutoBox) and data acquisition system, with the ability to execute embedded software directly generated from Simulink models.

+ General-purpose platform widely recognized in industry and academia.
+ Full integration with MATLAB/Simulink.
+ Already in use at CAR (EcoCAR teams)
– No specific support for diesel engine controls development.
– Very high cost relative to other options.

ETAS FlexECU
Open ECU development platform derived from production-intent Bosch engine control hardware. Supports embedded code execution from Simulink models, and provides a Simulink blockset for interfacing with system hardware.

+ Proven hardware platform
+ Full integration with MATLAB/Simulink.
+ Full integration with other ETAS tools already owned and used at CAR for various industry projects.
– Requires complete development of diesel engine control system structure, and operating maps.

Figure 3.2: Comparison of ECU platforms evaluated for RP-ECU project, with pros and cons.

The ETAS FlexECU was selected due to the significantly lower cost and compatibility with existing tools and workflows in use at OSU CAR.

3.4 Definition of control requirements

With the control system selected as above, the major task was the creation and implementation of a diesel engine control system structure. This section gives a summarized description of the development steps for the engine controls. The procedure below is expanded upon and generalized in Section 5.2.1, with the inputs and deliverables defined for each stage of the process.
3.4.1 Basic diesel engine control structure

Given the engine architecture described in 3.2, it was necessary to develop the following airpath and fuel controllers:

- Rail pressure controller
- Intake manifold pressure controller (variable geometry turbine controller)
- Fuel injection controller
- EGR controller

These controllers each required the development of sensor and actuator interfaces. A general description of how controllers drive sensor/actuator interface requirements is in 5.2.1.

The control structure was defined by generating a block diagram that described the inputs and outputs of each control loop (see Sec. 5.2.1).

3.4.2 Safety checks and diagnostics

In order to reduce the risk of accidental damage to the engine or the test cell, a system of software safety checks were incorporated into the control design.

Experienced engineering staff were consulted for suggested safety checks, and a short failure mode effects analysis (FMEA) procedure documented the failure modes that were discussed.

With these suggestions and analysis, the following safety checks were implemented:

- Engine coolant temperature max/min detection
- Engine overspeed detection
• Oil pressure loss detection
• Fuel rail overpressure detection
• Engine reverse direction detection

3.5 Controls development

3.5.1 Required components

Section 3.4.1 details the required development tasks for a new engine project. Following this outline, it was necessary to develop:

• Sensor and actuator interfaces
• Controllers
• Low-level software (LLSW) calibrations
• Controller calibrations

General information about components required for a new engine is detailed in Sec. 5.2.1.

The development of these components proceeded in parallel as described below.

3.5.2 Sensor development

Each sensor required by the control structure is defined by the physical sensor type (RTD, Hall-effect, digital on/off, etc), and the associated control loop input. Each sensor required software interfaces to be developed in order to allow it to be used with the control software. In general, development of sensors follows the procedure outlined in Sec. 5.2.2.
Example: coolant temperature sensor

As an example, the engine is equipped with a coolant temperature sensor. The coolant temperature sensor is a resistive temperature device (RTD)-type temperature sensor installed in the engine block that measures the temperature of the engine coolant.

Electrically, the coolant temperature sensor was connected to a physical input channel of the FlexECU appropriate to the sensor type and measurement frequency. A transfer function was calculated for the sensor that converted from the electrical signal to the output type specified earlier.

The sensor was validated by running the sensor block on the controller and verifying that its output matched a known temperature sensor over the sensor’s operating range. Once the sensor interface had been created and validated, it was added to the sensor interface library and merged into the master branch available to all users.

3.5.3 Actuator development

Similarly, each actuator (physical output) in the control structure needs to have an actuator interface created. Each actuator is defined by the actuator characteristics (solenoid, servo, DC motor, etc), and the associated control-loop output from Sec. 3.4.1.

Actuators were developed one at a time by the workflow outlined in Sec. 5.2.2.

Example: throttle actuator interface development

The engine’s throttle actuator serves as an example. The throttle actuator was defined by the relevant controller output (throttle position, in degrees) and the actuator physical type (DC motor with opposing nonlinear spring). This actuator required a feedback control loop inside the actuator block, so a simple control structure was designed (PI with position feedback).
The actuator block structure was implemented in Simulink and implemented in a test-bench model. This model was flashed to the ECU for calibration, and the feedback controller constants were tuned experimentally using the calibration software. Once the feedback control constants were finalized, the actuator interface was validated by observing the throttle response to reference inputs. The actuator block was completed by adding it to the library and merging it into the master branch available to all users.

3.5.4 Controller development

The heart of the control software is the controller models that define the control behavior. Controllers are usually developed in Simulink against an available engine model, using model-in-the-loop (MIL) techniques. For this project, an industry-standard control structure was used. Key calibrations (lookup tables/maps) for the controllers were provided by GM to speed the development process.

Example: manifold pressure controller (VGT controller)

The manifold pressure controller was defined by the control structure from Sec. 3.4.1.

A controller structure was designed by following a conventional VGT controller structure (feed-forward component and feedback component, with gain scheduling).

A test harness model was created to simulate the controller behavior against an engine model. During the MIL development process, the feedback controller was tuned by setting controller gains to meet simulation performance targets.

Once the controller had been developed using the MIL technique, it was finalized by putting it into a portable model format as described in Sec. 3.6.1 and 4.2.3.
3.6 Integration and validation

3.6.1 Moving controller models from simulation to ECU platform

After a controller model is developed in MIL simulation, some changes must be made in order to prepare the model for code generation and execution on the RP-ECU. For example:

- Table data for lookup tables and maps must be resampled to fit in the available data memory of the controller hardware
- Continuous-time controller models must be converted to discrete-time models at a fixed sample rate
- Some Simulink blocks used for debugging during the MIL process must be removed or converted to a type compatible with code generation

As a part of the project, a set of modeling guidelines, model standards, and guidelines were created to aid the developer in moving MIL models to the RP-ECU controller.

Rate selection for control loops

MIL development is often done with a continuous-time model and a variable-step solver to improve the speed and accuracy of simulation results. On the RP-ECU hardware, control models run as discrete-time controllers with a fixed rate. In order to move these models to the RP-ECU system, a fixed rate must be selected.

The RP-ECU platform provides a number of fixed-rate options for controllers (for example, 1ms periodic, 2ms periodic, 20ms periodic). Some synchronous rates (executed once per crankshaft resolution) are available.

Rate selection is done by considering the relevant physical time dynamics of the plant and the computational complexity of the controller. Because the ECU hardware has limited
computational resources, the control loop should be executed at the slowest acceptable rate. An acceptable rate is one that is well above the frequency of the relevant physical dynamics of the system.

**Example: intake manifold pressure controller**

The intake manifold pressure controller was developed as a continuous-time model using MIL techniques. In order to prepare the controller model for code generation and execution:

- The model was inserted into the software as a periodically executed subsystem, at a rate of 10 ms.
- Lookup tables in the model were downsampled from 1000 x 1000 to approximately 10x10 points (100 words). This did not result in an unacceptable loss of accuracy. More memory can be allocated for more complex tables.
- Disallowed blocks from the modeling guidelines were removed from the controller model.
- The solver configuration was updated to match the modeling guidelines (fixed-step solver with sample rate matching the execution frequency).
- Calibrations and parameters (see Sections 4.2.5 and 4.2.3) were added to the model to allow for calibration and debugging in the dynamometer test cell.

The model preparation step is a one-time process. After the model is prepared for code generation, it can still be used in MIL development, and modifications can be made in MIL simulation to update the RP-ECU target.

After this process is complete, the modified controller model was updated in source control and merged into the master development branch for execution and validation.
3.6.2 System-level validation

In order to begin system-level validation, the major independent development tasks were completed:

- Sensor and actuator interfaces were specified, designed, and validated
- Electrical connections were made between the RP-ECU, the engine, and any necessary sensors and instrumentation
- The control loops were developed against specifications or using MIL techniques, and prepared for code generation

After these tasks were complete, system-level integration and validation began. System-level integration is the process of combining controllers, sensor interfaces, and actuator interfaces on the RP-ECU hardware to achieve the system’s functional goals.

Debugging is more simple if the control loops are enabled and validated one at a time, in order of dependency. In the setup of the first project engine at OSU CAR, the following order was specified for integration:

1. Fuel rail pressure controller.
2. Fuel injection controller.
3. Intake manifold pressure controller.
4. Exhaust-gas recirculation (EGR) controller.

For each of these control loops, the controller was compiled and flashed to the RP-ECU platform. To validate the system, key measurements should be observed and recorded at a
number of steady-state operating points, and the controller behavior should be compared to the MIL simulation and/or the functional requirements.
Chapter 4: Software Architecture

Most of the utility of the RP-ECU system comes from the ability to create and modify ECU software as necessary. In order for the system to be useful, then, it should have a clearly defined and helpful method for storing software information and updating it as necessary: a "software architecture".

The software architecture should take into account the types of information that will be stored, and provide a sensible place for each of them. It should support the typical tasks that are undertaken during each stage of the software development process, and allow for those tasks to be completed in an efficient, straightforward way. And it should integrate well with the software interfaces of the tools in use today, as well as providing reasonable accommodations for unknown future tools.

This chapter describes the software architecture implemented for the RP-ECU system.

4.1 Problem overview: software architecture goals and requirements

4.1.1 Functional requirements

At minimum, the software architecture must:

1. Provide a set of tools that can be used to implement a broad range of engine controls on a target hardware platform.
2. Define a structure for storing the information generated in the process of ECU software development (code and data).

3. Provide a method for real-time debugging of the controls while they are running on the target hardware. Debugging is the ability to observe and modify data in the memory of the ECU.

4. Provide a method for capturing and recording data from the ECU for comparison or analysis, and exporting this data into commonly-used tools like MATLAB.

4.1.2 Goals

At a high level, the following goals were set:

- Portability: It should be possible to re-use subcomponents of the software for different purposes, and it should be possible to integrate software developed elsewhere without large effort.

  For example, a temperature sensor interface developed for a gasoline engine should be re-usable for a diesel engine without modification.

- Modularity:
  - It should be possible to develop the software subcomponents in parallel, so that multiple contributors can develop the software simultaneously.
  - It should be possible to replace one of the software subcomponents without requiring changes to other, unrelated parts of the software.

4.2 Solution overview: software architecture

The objective of the ECU software architecture design is to fulfill the goals outlined in Section 4.1. Broadly, the design does this by defining classes of information that must be
stored; defining a template structure in which they will be stored; and defining procedures for creating new software and modifying components of existing software.

When software development occurs using this development workflow and software structure, the development work products are well-suited for re-use across multiple platforms, and development work can progress rapidly in parallel.

### 4.2.1 Types of information

The ECU software has two major software components: **application software**, or ASW; and **base software**, or BSW. Each software component is made up of code (executable instructions) and data.

BSW originates with the ECU platform vendor, and is provided in compiled form along with the ECU hardware.
Figure 4.1: The four classes of software information that make up the ECU software, and the origin of each component. Application software (ASW) code and data is generated from a Simulink model of the controller. Base software (BSW) information is supplied by the ECU platform vendor, and modifications to the BSW data are stored early in the development cycle.

4.2.2 ASW software structure

ASW is the only component of the ECU software that the developer controls completely. The BSW is provided by the ECU platform vendor; although its data can be modified, its source code is not accessible and its behavior can only be changed in limited ways.

ASW is developed as a set of Simulink models, which are compiled into executable object files and combined with the BSW during the build process. The structure defined here applies to the ASW model files, which are analogous to the ASW "source code".
ASW hierarchy

The ASW model is a single model file, containing some number of independently operating control loops which correspond to the control loops identified during the engine control design process\textsuperscript{1}. Each control loop is executed independently and asynchronously at a specified rate.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{asw_hierarchy}
\caption{The hierarchy of the ASW model. A single container model contains one or more control loops, each of which operates independently and asynchronously. Each control loop is broken down into sensors, controllers, and actuators.}
\end{figure}

\textsuperscript{1}See Section 5.2.1 for a description of the development process for a new engine.
Each control loop is composed of sensor interfaces, controllers, and actuator interfaces. Clear separation between I/O (sensors and actuators) and controllers improves the portability of controllers and makes them easy to develop using a model-in-the-loop (MIL) workflow.

Together, the sensor interfaces and actuator interfaces form a hardware abstraction layer, which separates controller development from physical implementation.

The following sections define the structure and subcomponents of each ASW component.

### 4.2.3 Controller specifications

![Controller models in context of the ASW software hierarchy.](image)

**Figure 4.3:** Controller models in context of the ASW software hierarchy. Controllers are composed of a Simulink model, an initialization script, and one or more data files.

Controllers make up the heart of the ASW functionality. Each control loop has one or more controllers, which are developed independently of the overall ASW.

Each controller is stored in a controller package, which consists of three subcomponents:

- a controller model,
- an initialization script, and
Controller model specifications

The substance of the controller algorithm is specified by the controller model. The controller model is a single Simulink model file with root-level inports and outports corresponding to the inputs and outputs (arguments and returned values) of the controller algorithm.

Controllers are included in the master model by model referencing. Separating each controller into a separate model file improves the utility of file-based version control, since changes to a single controller only affect a single file on disk.

Each controller must explicitly define any signals that should be available for measurement or datalogging purposes, as well as the data that should be made editable during the calibration process. These definitions are made in the controller model itself and the model initialization script, respectively.
**Initialization script**

The model initialization script defines the model parameters that should be editable at runtime, and loads any model data that is stored in .mat or other external file types.

The controller model executes the initialization script automatically when it is loaded, during the `PreLoadFcn` model callback.

Parameters are software variables that do not normally change at runtime\(^2\). During the controller development process, the developer usually wants to edit the value of some parameters to observe the effect on the software: for example, the proportional gain of a PID controller is a software parameter, and a developer might want to modify its value several times during the process of tuning the controller.

The model initialization script defines `Simulink.Parameter` objects in the base MATLAB workspace with the `DataType` attribute set to `ImportedExtern`. `Simulink.Parameter` objects defined in this way will automatically be made editable in the calibration software (INCA).

In summary, the model initialization script must:

- load model data from any external data files
- define `Simulink.Parameter` objects corresponding to data that should be editable during the calibration process

An example of a model initialization script is provided in Appendix ??.

**Data files**

If the controller depends on a set of data files (for example, lookup tables) to operate, the data can be stored in a set of external files for convenience. Typically, the data will

\(^2\)See Section 4.2.5 for a more detailed definition of parameters.
be stored in one or more .mat files. The data is loaded into the base workspace by the initialization script.

4.2.4 Sensor and actuator interface specifications

Sensor blocks provide a physical measurement to a controller. They have no time-varying inputs in the modeling tool, although their internal code accesses hardware inputs on the control unit. For example, real-time measurements from a manifold air pressure sensor is accessed using a sensor block.

Sensor blocks are stored in a Simulink library file, and must be completely self-contained. They are not permitted to require a separate initialization script or data store in order to operate, and it must be possible for multiple instances of the same sensor block to run simultaneously in the same model file.

4.2.5 ASW interfaces

Run-time software interfaces

While the ECU software is running, the ASW code’s only external interface is with the BSW.

Sensor interfaces receive data from BSW interfaces: for example, many sensor interfaces receive data from a BSW function that provides the most recent measured value from an analog-to-digital converter (ADC).

Actuator interfaces send data using BSW interfaces: for example, a throttle actuator interface might cause a certain current to flow by adjusting the PWM duty cycle of an output pin using a BSW function. Some actuator interfaces also read data from the BSW.
Calibrations and measurements

The behavior of the ASW can be observed and adjusted, respectively, by using measurements and calibrations.

![Diagram of control loop 1 with sensor, actuator, controller, measurement 1, measurement 2, calibration tool, calibrations, and measurements]

**Figure 4.5:** Controllers expose measurements and calibrations to the developer, which are accessible using the calibration tool. Measurements are read-only, and allow observation of variables inside the ASW. Calibrations allow data inside the ASW to be altered at runtime.

**Measurements** Measurements are variables that are defined in the modeling tool as being observable at runtime\(^3\). The developer can use the calibration tool to observe and record the current value of a measurement in real time.

---

\(^3\)See section 10.5.1, "Creating Model Measurements and Calibration Data", of the EHOOKS-DEV V3.0 User Guide for information on defining measurements in Simulink models.
Calibrations, parameters  Calibrations, or parameters, are variables that are defined in the modeling tool as being editable at runtime\(^4\).

Some calibrations may allow for the adjustment of a controller by changing constants or tables used by the algorithm: for example, the table data for a lookup table might be designated as a calibration so that the table’s output could be changed at runtime.

Other calibrations might serve as switches to enable or disable certain software components: for example, a controller might include a feed-forward and feedback component, whose outputs are either included or not included in the controller output. A calibration could be used so that the developer could enable or disable each component without re-compiling the ASW entirely.

\(^4\)Calibrations are defined in the controller model initialization script - see Section 4.2.3.
Chapter 5: Workflow and toolchain

At a high level, the rapid prototyping system exists to support the process of engine controls development. This process often involves tasks like:

- controller implementation: adding a new control loop to existing ECU software, sometimes using additional sensors and actuators
- controller calibration: observing the behavior of a control loop under certain experimental conditions, and modifying controller data to alter the controller behavior
- engine characterization: exercising the engine across a range of operating conditions, and recording data about its behavior and response to stimuli

These tasks involve a variety of software tools, each of which is used to interact with a different component of the system. The "Toolchain" section of this chapter outlines the different tools that are used throughout the development process, as well as the flow of information between tools throughout the development process. The "Workflow" section presents several common tasks and outlines how they are accomplished using the software tools.

5.1 Toolchain

The developer interacts with the system using software tools. This section defines the four major software tools used and describes the flow of information between the tools throughout the development process.
5.1.1 Core tools

- The **modeling tool** is used to develop the application software (ASW), which is composed of the controllers and the sensor/actuator interfaces. It defines the behavior of the ASW and the data that will be visible for measurement or calibration. In this project, Simulink is used as the modeling tool.

- The **calibration tool** is used to edit data used by the ASW and the LLSW. After the behavior of the ASW is defined using the modeling tool, the calibration tool is used during the software integration and validation process. It is usually able to update the calibration data after the software has been compiled, so that the user can modify the behavior of the software without needing to re-compile it. In this project, ETAS INCA is used as the calibration tool.

- The **build toolchain** is the set of tools used to create executable code from the model, compile that code into executable instructions, and integrate it with other compiled code (the base software, or BSW). This project uses ETAS EHOOKS software along with the Simulink Coder code-generation package.

- The **version-control tool** is a tool used to track changes made to the ECU software. It maintains a verifiable log of all changes, and provides ways for multiple developers to collaborate on the ECU software independently.

5.1.2 Data flow

Compilation and flashing

Flashing is the process of transferring the ECU software onto the physical ECU hardware, so that it can be executed on the ECU for calibration or testing. Before the hardware can be flashed, the ECU software must be compiled from the various files that store its
components. Figure 5.1 shows how information flows between the various software tools during the compilation and flashing process.

**Figure 5.1:** Flow of information between the various software tools during the software compilation and ECU flashing processes.

### 5.2 Workflow

#### 5.2.1 Development of ECU controls for a new engine.

When an engine controller is needed for a new model of engine, several development tasks are required to prepare the engine controller. This section outlines the process of developing ECU software from the ground up.

A useful engine controller requires:
• Controllers: a set of software functions that modify the output of physical actuators in response to signals from sensors.

• Sensor and actuator interfaces. Most signals that are input to controllers come from sensors, and each sensor requires a sensor interface that maps its electrical signal onto the corresponding physical value being measured. Most controllers’ outputs are sent to actuators, which cause a physical change in response to a command. Each actuator requires an actuator interface, which abstracts away the electrical implementation of the actuator behavior.

• Low-level software (LLSW) calibrations are required to configure the functionality provided by the ECU. For example, the number of cylinders in the engine, the firing order, and the format of the signals from the engine’s position sensors need to be defined in order for the LLSW to provide engine speed measurement and injection management functions.

• Controller calibrations: data that drives the behavior of the ASW controllers. For example, if an engine’s fuel injection controller includes lookup tables for the quantity of fuel injected, the table data is an engine calibration.

In order to develop the components above, the following procedure can be followed:
1. Identify control objectives and plan control structure.

2. Identify sensors and actuators necessary for control structure.

3. In parallel:
   
   • Develop sensor and actuator interfaces.
   
   • Develop controllers.
   
   • Develop LLSW calibrations.

4. Integrate controllers with sensor and actuator interfaces one by one.

The following sections describe each of the above steps in more detail.
Identifying control objectives and planning control structure.

**Inputs**  |  Engine platform information, project engineering objectives
**Deliverables**  |  Flow diagram of controls, with defined inputs and outputs

The engine platform being targeted, and the engineering objectives of the development project, will define some set of controllers that must be implemented. For example, all electronically controlled fuel-injected engines will require a controller that determines the timing of the fuel injections (i.e., the moment at which each injection should begin, and the duration of each injection.) Most Otto-cycle engines will require a controller that specifies the position of the throttle; a forced-induction engine requires a controller that specifies the intake manifold pressure target. These engine platform features, as well as the project engineering objectives, are the inputs to the process of defining the control structure.

The control structure is defined by a block diagram listing each of the controllers, their inputs and outputs, and how they are connected. The inputs and outputs are defined in terms of their physical values. For example, a fuel injection controller may be defined as in Figure 5.3.

![Figure 5.3: Example of a single controller with inputs and outputs. The inputs and outputs are defined in terms of their physical units.](image)

Figure 5.4 is an example of a completely defined control structure with four controllers.
Each controller in the structure might be subdivided internally into several additional components, but defining the subcomponents is not necessary at this stage.

![Figure 5.4: Example of a top-level control structure with inputs, outputs, and controllers.](image)

Once the control structure is completely defined, the task is complete.

**Identifying sensors and actuators**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Block diagram of controls generated from previous step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverables</td>
<td>list of all top-level inputs and outputs (I/O); mapping between top-level I/O and sensors/actuators</td>
</tr>
</tbody>
</table>

The control structure (block diagram) defines signals that are required by a controller but not supplied by any other controller (top-level inputs), and signals that are supplied by
a controller to an external entity (top-level outputs). In order to plan out the development of the hardware abstraction layer (sensor and actuator interfaces), this step involves listing each of these top-level inputs and outputs, and defining their physical source or destination.

A top-level input might be supplied by a physical sensor: for example, the intake manifold pressure might be provided by an electronic pressure sensor. Or it might be supplied by the LLSW: the engine speed, for instance, is a signal that is supplied in this way (it is calculated automatically by the LLSW). Or it might be a system-level input provided by the user, like the torque command. Each of these input signals should be listed along with its data source. If the data source is a physical sensor, the sensor should be specified. Table ?? is an example of the input signal definitions for the control structure shown in Figure 5.4.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Source</th>
<th>Physical sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel rail pressure [kPa]</td>
<td>Fuel rail pressure sensor</td>
<td>Fuel rail pressure sensor</td>
</tr>
<tr>
<td>Engine speed [rpm]</td>
<td>LLSW engine position driver</td>
<td>User input</td>
</tr>
<tr>
<td>Engine torque command [Nm]</td>
<td>User input</td>
<td></td>
</tr>
<tr>
<td>Intake manifold pressure [kPa]</td>
<td>Intake manifold pressure sensor</td>
<td>Intake manifold pressure sensor</td>
</tr>
</tbody>
</table>

Table 5.1: Input and sensor specifications for top-level diagram shown in figure 5.4.

A top-level output is usually sent to a physical actuator, like a throttle actuator, or an EGR valve. Alternatively, a top-level output might be used only for analysis or user feedback. Like the top-level inputs, top-level outputs should be specified along with their destination, and if the destination is a physical actuator, that actuator should be specified.
Table 5.2: Output and actuator specifications for top-level diagram shown in figure 5.4.

These two tables define the sensor and actuator interfaces that need to be developed in Step 3.

Develop sensor and actuator interfaces

Inputs

- List of all sensors and data sources
- Interface specifications (electrical characteristics, etc.) for all sensors
- List of all actuators and data sinks
- Interface specifications (electrical characteristics, etc.) for all actuators

Deliverables

Simulink library with sensor and actuator interface blocks

Each unique sensor and actuator specified in Step 2 must have a sensor interface, or an actuator interface, developed for it. These interfaces separate electrical implementation
from control design, allowing sensors to be modified as necessary without requiring changes
to control design, and allowing the development tasks to occur in parallel.

Most sensor interface development will be based on the sensor specifications provided
by the manufacturer, and most sensor interfaces are simply a mapping of the sensor’s
electrical output onto the physical value being measured. For example, the sensor interface
for a resistive temperature sensor, or RTD, measures the analog-to-digital converter (ADC)
channel assigned to the RTD, then uses an inverse thermistor equation to convert the
measured voltage into a temperature value.

![Coolant temperature sensor interface]

Figure 5.5: Data flow inside a sensor block. The physical value lookup
is shown here as a 1-D lookup table, but it might also be a mathematical
equation or more complex mapping.

Sensor interfaces can be more sophisticated: for example, a sensor interface might in-
corporate a low-pass filter or estimator to process a noisy signal; or the interface to a digital
sensor might include functionality to interrogate the sensor and interpret its response. These
more complicated sensor interfaces have not been necessary to date.

In general, sensor blocks are developed and validated by testing them against the phys-
ical sensor hardware. Most sensor interfaces can be validated by running them on the ECU
inside a barebones control model and observing their output.

For more detailed specifications on the structure of sensor interfaces, see Section 4.2.4.
The workflow for developing a new sensor/actuator interface using version control is described in Section 5.2.2.

**Develop controllers**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>List of controllers from system block diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverables</td>
<td>Controller packages (model, data, initialization script) for each controller</td>
</tr>
</tbody>
</table>

Controller models are the core of the software functionality. Their design process depends somewhat on the control strategy chosen to implement their behavior; as such, this document does not specify their internal structure in detail.

In general, controllers are developed against a simulated engine model when possible. The software structure of the controllers is designed to facilitate this. Each individual controller is a standalone Simulink model file with top-level inputs and outputs (Simulink Inport and Outport blocks). The controllers are included in the ECU software using Model Reference blocks.

In order to develop the controllers before the sensor interfaces are available, a model-in-the-loop (MIL) development process can be used if a suitable engine model is available. The controller under development is included in a larger engine model, which can be used to test its behavior in simulation. Once the controller behavior is acceptable in the simulation, it can be used as-is in the ECU software.

Section 5.2.2 describes the process for developing and adding a new control loop using the version control tool. This process can be followed repeatedly to add each new control loop under development. Multiple developers can work on different controllers in parallel without needing to share any files.
 Inputs | engine sensor specifications and engine parameters (firing order, cam and crankshaft signal descriptions, etc)  
Deliverables | Data file with validated LLSW calibrations

**Develop LLSW calibrations**

Some important functionality is provided by the low-level software (LLSW) supplied with the ECU. Notably, the fuel injector driver, as well as the sensor interface for engine position and speed, is provided by the LLSW. These functions need to be configured for the engine platform by setting certain calibration values.

The procedure for developing these LLSW calibrations is described in the ECU documentation. Since the procedure provided in the ECU documentation is completely inadequate, another, better procedure has been developed for users of the RP-ECU project. Validation of the LLSW functions can be performed on the dynamometer using calibration software.

**5.2.2 Collaboration using distributed version control system (DVCS)**

The RP-ECU system is developed to be used with file-based distributed version control software: specifically, a software package called Git, originally created by the developers of the Linux kernel to facilitate collaboration on that large, complex software project by a diverse, geographically distributed team.

Using version control software has many benefits. For this particular project, the most notable benefits include:

- A log of all of the changes made to the software over time, including the ability to restore a historical version of any file as needed
- A system for managing development work separately from a "master" copy of the software, which remains intact until the changes to be made are finalized
A system that allows several developers to work on new features in parallel without affecting the master copy or one another, including a formal process for merging all of the individual work products when they are complete.

Git provides these benefits without requiring all work to happen on a central server; individual developers working on the project can make changes on their local computers, and synchronize them as necessary.

Because many controls engineers have not worked with a DVCS before, this section will provide a brief overview of the terminology and concepts. A number of example workflows are given that cover the most common use cases in the RP-ECU project.

**DVCS fundamentals**

The function of the DVCS (Git) is to track and manage changes that are made to a set of files over time. Many guides focus on Git features from a software implementation perspective; this section will describe only the behaviors that are useful in the controls development process.

The ECU software is stored in a single Git *repository*. A repository is simply a folder whose contents are tracked by the DVCS. The ECU software architecture defines a folder structure within the repository, and specific files that must exist, but Git is able to track any file in the folder.
Terminology

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>The folder containing the ECU software (ASW, BSW, calibrations, data, and other configuration files).</td>
</tr>
<tr>
<td>working copy</td>
<td>The working copy is the repository as it currently exists on the user’s computer. The working copy may include changes to files in the repository that have not been saved in source control yet.</td>
</tr>
<tr>
<td>Commit (noun)</td>
<td>A single snapshot of the repository at a moment in time.</td>
</tr>
<tr>
<td>Commit (verb)</td>
<td>To save changes to files in the repository by creating a new commit. The user can select which files they want to commit and write a brief message describing the changes that have been made.</td>
</tr>
<tr>
<td>Branch (noun)</td>
<td>A branch is a copy of the repository that is used to develop a feature without disturbing the other copies. Branches can be created from any commit, in order to develop independently. The user merges two branches in order to integrate the changes once they have been made.</td>
</tr>
<tr>
<td>Branch (verb)</td>
<td>To branch is to create a new branch based off an existing commit.</td>
</tr>
<tr>
<td>Merge (verb)</td>
<td>To integrate changes from a different branch into the current branch.</td>
</tr>
<tr>
<td>Remote (noun)</td>
<td>A remote is a complete copy of the Git repository hosted on a server. Remotes are used to back up and share work between multiple developers.</td>
</tr>
<tr>
<td>Push (verb)</td>
<td>Upload any new commits from the current branch on a local computer to a remote.</td>
</tr>
<tr>
<td>Pull (verb)</td>
<td>Download changes from a remote branch into the current branch.</td>
</tr>
<tr>
<td>Reset (verb)</td>
<td>Discard changes to a file, or all files in a repository, since a certain commit. Resetting effectively restores the file(s) to their state at some point in the past.</td>
</tr>
</tbody>
</table>

Table 5.3: Common Git terminology for software developers.

Workflow example - adding a new sensor/actuator interface.

Sensor and actuator interfaces are stored in a single Simulink library file, rp_ecu_library.slx. When a controller is added to the ASW, new sensor and actuator interfaces may need to be developed to supply its inputs or realize its outputs.

Before the interface is merged into the master library, it should be validated on the ECU to verify its functionality. This requires an ASW model that can be used to flash the interface onto the ECU hardware and observe its behavior. Instead of creating such a model from scratch, the developer can use the existing ASW model as a scratch workspace, and then discard changes to that file before merging the development branch back into the
Workflow example - adding a new control loop.

In order to add a completely new control loop to the ASW:

1. Branch from the master.

2. Add the controller files (model, data, initialization script) to the Controllers directory, and commit the added files.

3. Add the control loop to the ASW model:
   (a) Create a new function-call subsystem at the top level of the ASW model.
   (b) Specify an appropriate trigger source based on the timing requirements of the controller.
   (c) Add sensor and actuator interfaces as necessary from the RP-ECU library. Add the controller as a referenced model.

   Commit the changes to the ASW model.

4. Validate the behavior of the modified ASW, make changes as necessary, and commit the finalized model.

5. Push the development branch to the remote. File a merge request against the master branch.

   The maintainer of the project will review the merge request against the project merge guidelines. Once it is accepted, the master will include the newest control loop.

5.3 Documentation

In order to streamline the development process for future users of the project, and taking into account the need for effective knowledge transfer tools in a student research
environment, a set of documentation was created over the course of the project. The goal of the documentation task is to anticipate and answer time-consuming questions that arise during the development process.

To that end, a number of use cases were developed based on experience with the project, and a list of common questions was developed based on informal interviews with students who had worked on the project. After reviewing these common questions, a list of useful documentation deliverables was created.

This documentation includes:

- Software structure (abstraction layers - conceptual overview)
- Modeling guidelines for Simulink models, similar to MAAB guidelines
- Electrical wiring diagram with examples of implementation of common sensors
- Overview of FlexECU documentation and comments on poorly-written sections of manufacturer documentation
- Overview of version control software concepts and terminology
- Step-by-step workflow for prototyping new sensor/actuator interfaces using version control software
- Improved overview of FlexECU low-level software calibrations, with specific information on which calibrations are necessary for engine functionality
- List of required software tools and versions
- Documentation of virtual machine configuration and sample virtual machine for project work
- Step-by-step workflow for downloading changes from version control system
- Step-by-step workflow for uploading new changes to version control system
  Moving an existing controller

- Step-by-step procedure for flashing (programming) updated software onto the RP-ECU system.

- Dynamometer startup and shutdown safety procedures

- Fuel system startup/shutdown procedures

The documentation is stored in part on a central repository hosted by the University, called University Code Repository (UCR), in a wiki. UCR also hosts the version-control software used by the project, and allows for access to confidential information to be granted securely.
Chapter 6: Conclusions

After years of experience in developing embedded engine controls, the RP-ECU project was chartered to reduce duplication of efforts and allow for faster, less expensive controls development at Ohio State’s Center for Automotive Research. After about a year of active development, the project has yielded a functioning engine control system and is well on the way to system-level controls validation, all at a cost of less than half of commercially available "turn-key" systems.

Immediately after the completion of system-level validation, the first research project to be implemented on the RP-ECU platform will begin experiments. In the coming years, the RP-ECU system will be used for academic and industrial projects and continue to reduce cost and increase sophistication of development efforts at OSU CAR.
References
