Freescale Semiconductor
User's Guide

Document Number: KT33812ECUUG
Rev. 3.0, 11/2009

Small Engine Reference Design User Manual

Featuring the MC33812 and MC9S12P128

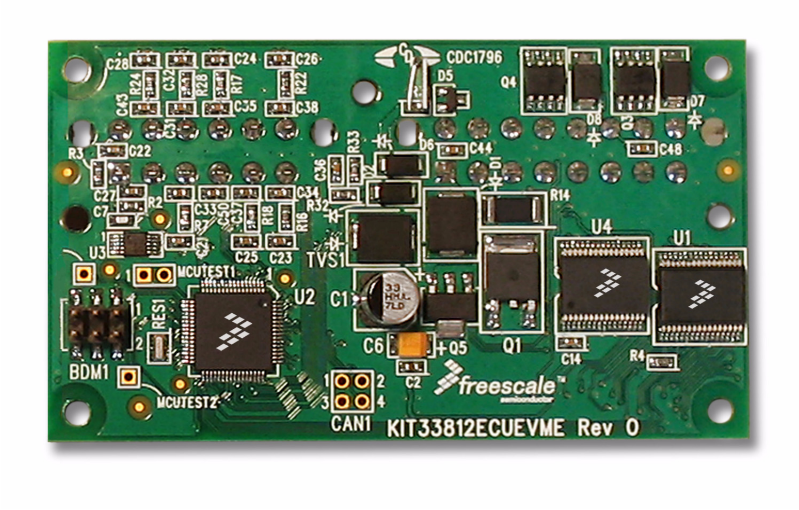


Figure 1. KIT33812ECUEVME Evaluation Board

Table of Contents

1 Introduction

Welcome to the Freescale Small Engine Reference Design Kit. This product was designed to be a complete solution for the electronic control of a small engine. Small engines are defined as a one or two cylinder engine for use in anything from a motorbike to a lawn mower to a generator. While the decision was made to address a one-cylinder engine specifically, this design is extremely useful for a two-cylinder engine with little or no modification. Freescale's concept of creating an engine control kit is intended to enable a market ranging from garage hobbyist to seasoned Tier 1 Powertrain Engineer using Freescale products.

Through the use of this kit, you can create an engine controller specific to a small engine application. Engine control is a discipline that requires intimate knowledge and experience in Chemical, Mechanical, and Electrical Engineering. For those familiar with mechanical control of an engine through a carburetor, the use of this reference design kit can help to advance your knowledge in the electrical area and provide a jump-start for a successful adoption of electrical engine controls to meet new emissions standards. Providing a kit such as this is intended to make semiconductor products from Freescale easier to use. The user is responsible for providing all input signals, output loads as well as the completed system design and development. This kit should serve as a starting point for the development of an application specific engine controller for a small engine. Example software and documentation are provided to assist in successful design and implementation. It is recommended to have the following skills and experience: embedded C-language programming, analog and digital circuit design and schematic analysis, microcontroller programming, fuel injection system debugging and calibration, and engine test environment experience. Additionally, there is further benefit to experience using the CodeWarrior Development Studio and the Freescale S12(X) microcontroller Units (MCUs). The User Reference Manual provides exercises and references to additional information to reduce the learning curve for inexperienced users.

Freescale's goal is to enable the small engine market. To clarify this point, the hardware included in this kit can readily be configured and reprogrammed to run an engine. However, it lacks the application specific hardening (EMC, ESD, and environmental areas for example) and implementation optimization that make it a production ready module for any specific application. Further, the free example application software provided is a starting point capable of running an engine. It does not apply any advanced control strategy capable of addressing the pollution concerns and regulations facing the small engine industry. To do this would become application specific to an engine and could not be and should not be implemented by a semiconductor supplier as it is deeply outside their area of expertise. The example application software does show how to use the key functionality in the Freescale products that the kit is based on, which speeds up the development process by showing a working example.

The contents of this kit will save many months of work, even for experienced powertrain engineers just looking to evaluate Freescale products. A system has been created based on a one-cylinder closed-loop engine controller using integrated technology while being cost-effective for the small engine market. Example software is provided that can be customized to run an actual engine that has electronic fuel injection. Documentation is provided to aid in going through the process of developing an application. Finally, information on modifying the design to support the adaptation of the small engine reference design to your application goals.

2 Getting Started

2.1 Exploring the Contents of KIT33812ECUEVME

Included in this kit are the essential components to develop an engine control application for small engines. Development is centered on the use of a Windows based PC and the Electronic Control Unit (ECU) contained in this kit. **Figure 2** shows a picture of the key kit contents. The key components of the kit are: ECU, wire harness, documentation DVD, Freescale CodeWarrior for the S12(X) (contained on DVD), USB BDM Tool, and USB cable. Please refer to the packing list for any additional components that may be included in the kit. If contents are missing, use the included warranty card or contact your local Freescale Support Team.

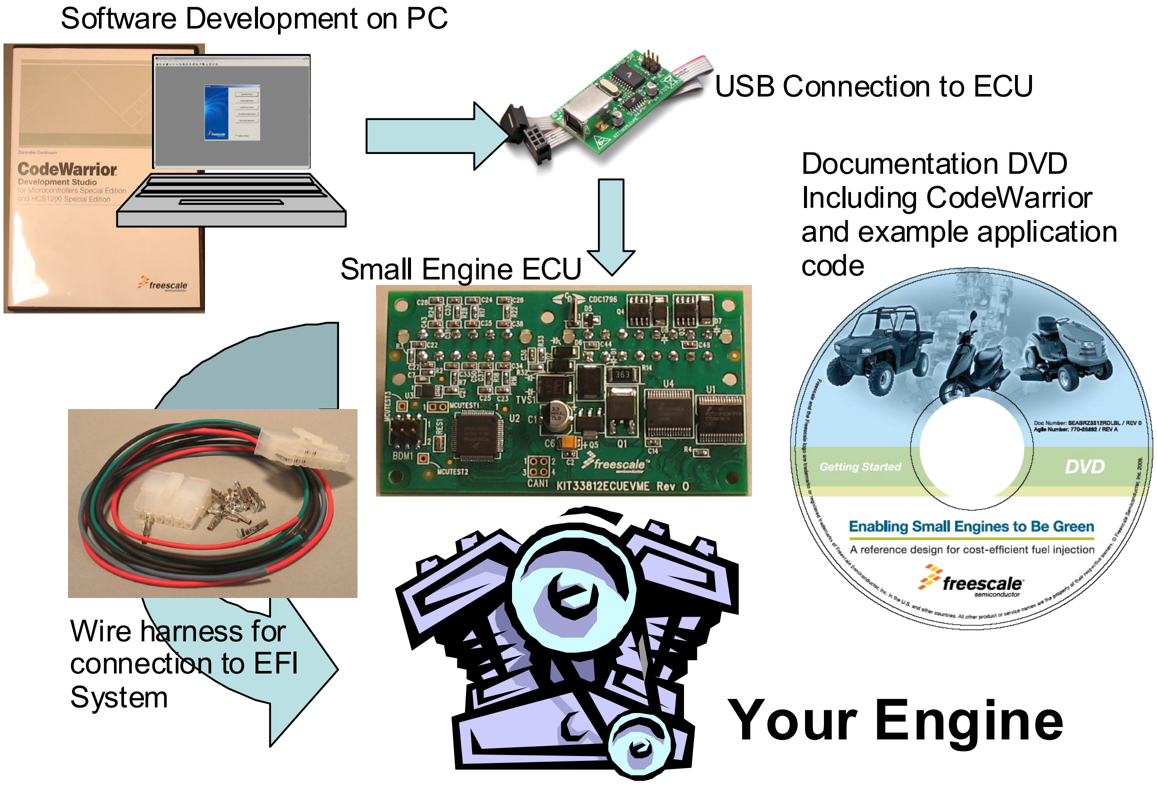


Figure 2. Contents of Kit KIT33812ECUEVME

2.2 Electronic Control Unit (ECU)

This is the Small Engine Reference Design hardware. It is a one-cylinder engine controller based on the Freescale MC9S12P128 microcontroller, MC33812 Small Engine Integrated Circuit, and MC33880 Configurable Octal Serial Switch. The unit will run from a 12 V battery and control engine loads such as a fuel injector, inductive ignition coil, relays, incandescent lamps, and LEDs. The ECU also takes inputs from switches and sensors, such as Engine Stop switch, manifold air pressure, engine temperature, and variable reluctance sensors. Application software will be run on this unit containing your engine control strategy. While the unit is not designed to be a production module specific to any engine, it is intended to have the same look and feel. This resulted the small, business card sized form factor and minimal provision for expansion.

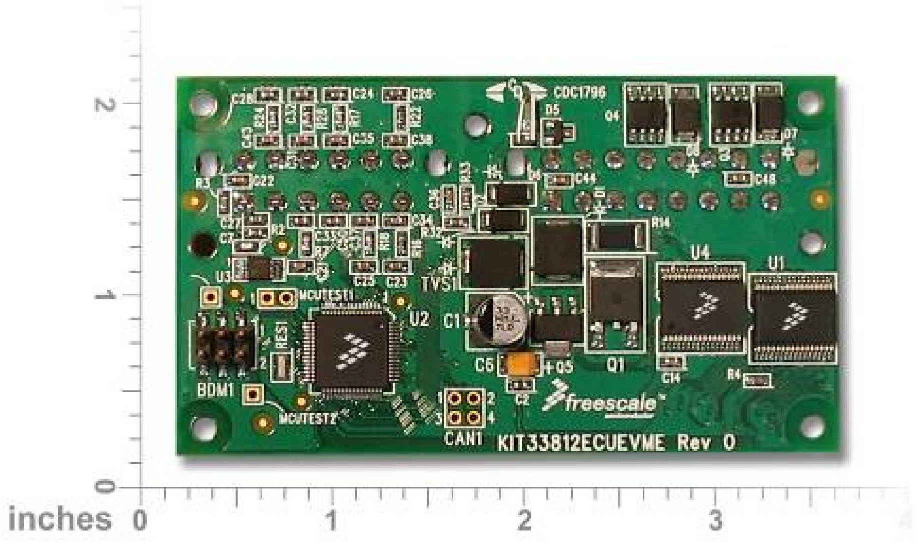


Figure 3. ECU Included in KIT33812ECUEVME

2.3 ECU Wire Harness

To provide a physical connection to the electronic fuel injection system, a wired connection to the controls and sensors of the system is required. As a starting point, a basic wire harness is included in the kit along with the components to fully populate the connectors. The basic wire harness allows power to be applied to the module and a minimal set of loads. Later in this manual, there is documentation that will discuss the process of interfacing the signals of the engine to the ECU. Addition connectors can be easily obtained through known electronic component supplies. Exact part numbers are made available in the bill of materials (BOM) for the ECU.

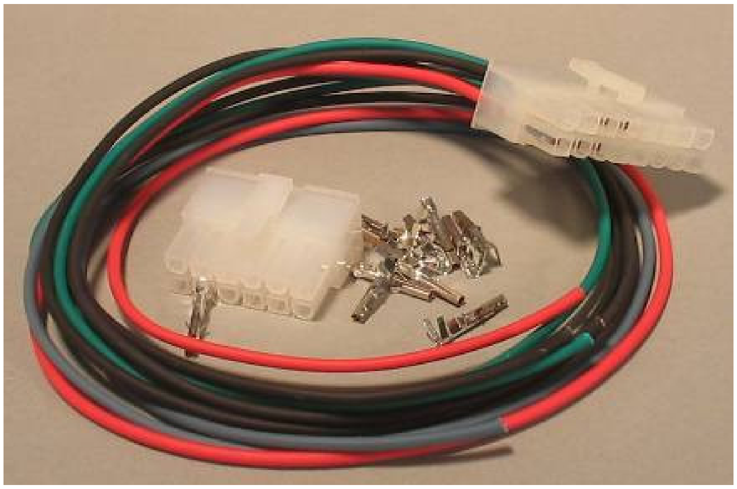


Figure 4. ECU Wiring Harness

2.4 Documentation DVD/CD

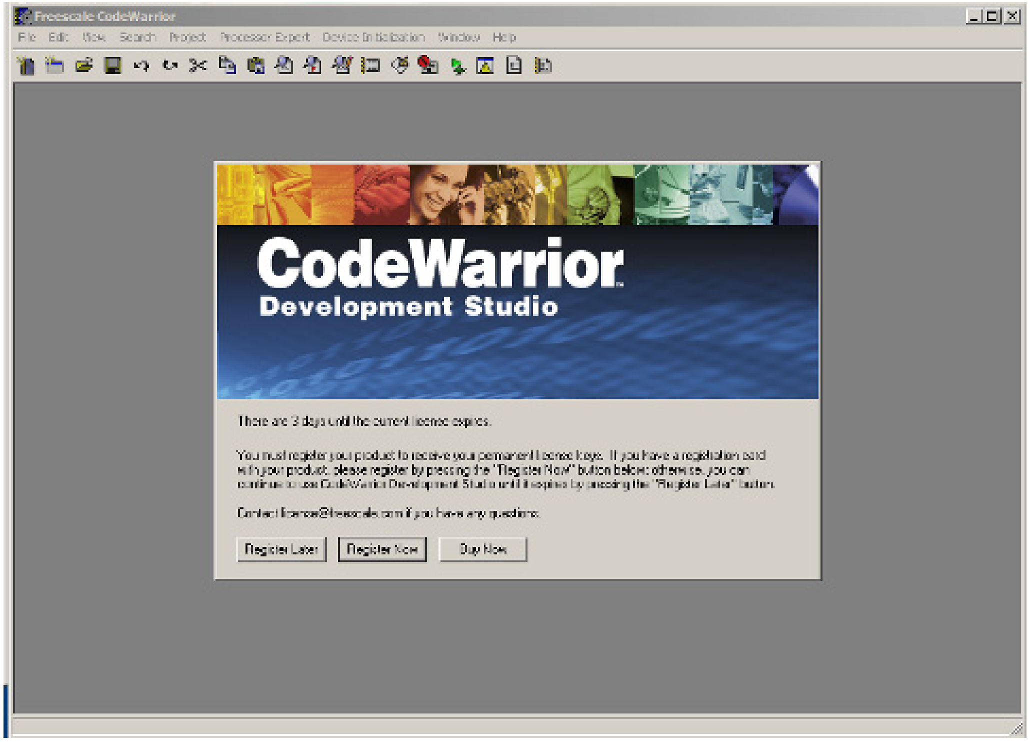
The documentation media contains electronic copies of all relevant information for creating and using this kit, including this User Manual. Documentation includes various support tools, such as spreadsheet tools, and design files including schematics and Gerber output files. These can be accessed through the graphical application that is automatically launched or by using Windows Explorer as a more direct navigation of the contents. As information may be updated, always reference www.freescale.com for the latest relevant information.

Getting Started
DVD
Enabling Small Engines to Be Green
A reference design for cost-efficient fuel injection
freescale
semiconductor

Figure 5. Small Engine Reference Design Documentation DVD/CD

2.5 Freescale CodeWarrior for the S12(X)

All software for the ECU is developed using this application, which is included on the DVD. This is done as a convenience as it is a large program to download. It is recommended to check for the latest version and updates at www.freescale.com. Example software is tested using the 5.0 release of the S12(X) product. This version does not require any updates or patches at the time development, however it is recommended to maintain this software through updates as available. The CodeWarrior Development Studio is an integrated development environment that provides a common interface for working with the various tools needed for building software. It comes in various levels of product for various types of MCUs. The example software allows the use of the Special Edition Product which is free for use. As your application grows and further features of the product are required, upgraded licenses can be purchased to meet your needs. The primary function of the CodeWarrior application is to compile software, program the ECU, and then control the execution of the software through the integrated debugger.



There are 3 days until the current license expires.

You must register your product to receive your permanent license keys. If you have a registration card with your product, please register by pressing the "Register Now" button below; otherwise, you can continue to use CodeWarrior Development Studio until it expires by pressing the "Register Later" button.

Contact license@freescale.com if you have any questions.

Figure 6. Screen shot of Freescale CodeWarrior for the S12(X)

2.6 USB BDM Tool

Connection from the Windows PC to the ECU is performed by the USB to BDM Tool. This tool is powered through USB and interfaces with the CodeWarrior application. The link to the ECU is through a 6 pin ribbon cable that goes from the USB BDM Tool to the Background Debug Module (BDM) header on the ECU. Through the BDM connection, the CodeWarrior application can use the BDM tool to communicate, program, and control the S12 microcontroller on the ECU. While tool gets its power from the USB port on the PC, it does not power the ECU. This separation is important as it provides a level of isolation from the engine system to the development PC. Initial kits may include the P&E Multilink as the USB to BDM Tool. Normal production kits will make use of the TBDML. It is important to know which tool you are using so that the proper connection is selected when using CodeWarrior.

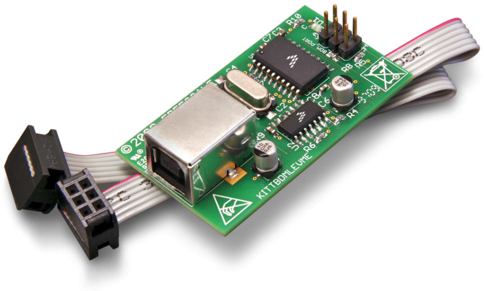


Figure 7. Example USB BDM Tool for Connection to PC

2.7 Additional Recommended Hardware

In addition to this kit, various pieces of equipment are recommended to perform application development work for software validation and testing. These are commonly found in most electronics labs:

- 12 V, 10 A DC power supply
- 100 MHz (minimum) 4 channel oscilloscope
- Soldering iron
- Grounded electrostatic matt
- Windows XP (required) PC
- 12 V relays
- Potentiometers
- Switches.

Having all of these items will allow testing and debugging of the system.

3 System Setup

Now that the contents of the small engine reference design have been described, the focus will shift to the complete development system. This includes the contents of this kit and the fuel injected engine as a system. At a high level, system setup contains the following steps:

- Definition of interface between ECU and Engine
- Creation of simulated engine environment
- Installation and verification of software development environment
- Engine load and sensor validation
- Migration plan towards real engine hardware

To accomplish these steps, several exercises will be described to help take you through this critical phase. These exercises include, getting started with Freescale CodeWarrior, and creating a known reference system. Additionally, suggestions for further training will be provided based on using Freescale products and the system level setup. **Figure 2** shows the components of this kit and a placeholder for your engine. This system incorporates the interface from the PC to the actual engine. The user must provide the engine loads for electronic fuel injection including fuel injector, inductive ignition coil, relays, and other relevant components. Signals from VRS, MAP, switches, and other inputs must also be provided along with the actual engine itself.

3.1 Definition of interface between ECU and Engine

The first step in using this kit is to determine how it will connect to your engine system. As mentioned, the engine must be fuel injected. If you are converting an engine from mechanical (carburetor) to electronic control, this must be done before or in parallel with using this kit. The ECU is designed around a one-cylinder engine, however, it can be adapted to work with a two-cylinder engine. The requirements for two-cylinder operation are: a) ignition coil must be a dual output or twin coil, b) wasted-spark strategy is acceptable for application. This means that a twin coil, capable of driving two separate spark plugs from a single input, can be used to fire every engine revolution (in a four-stroke engine) to produce two spark events, one in the desired cylinder and one in the second (wasted) cylinder. If this can be tolerated in the system, fuel control can be provided individually to each cylinder through the INJOUT and ROUT1 signals.

To aid in the connection from the ECU to the engine, a worksheet is provided. Using Load Worksheet.xls, available on the documentation DVD, connection to the engine can be defined. This Excel spreadsheet contains the full list of connections and suggested functionality for each pin of the ECU. Matching up the various controls, sensors, and inputs on the engine to the ECU should take into account voltage ranges and current capabilities. If there is doubt to the connection, use the schematic and Hardware Design Reference Manual found in Appendix A to for in-depth analysis of the circuits behind each ECU level pin.

The design goal of a cost-efficient design does not allow for a system to include all possible system configurations. The signals available reflect essential controls for one-cylinder, closed-loop engine control, highlighting the integration of the MC33812 Small Engine IC. Essential functionality should be considered first, such as the direct controls for fuel and spark. System controls such as the fuel pump or voltage regulator should be secondary concern as they can be externally controlled and do not require precise timing execution.

By filling in the information under the “Target Engine System” column, see yellow highlight in Engine Load Worksheet - Target System Identification Column in Yellow, each connection to the ECU can be defined. In the actual worksheet, signals of the ECU are color coded to identify similar functionality. From this completed worksheet, the wire harness from the engine to the ECU can be made. Materials for the AMP brand connectors of the ECU are included to get this process started.

Engine Load Worksheet - Target System Identification Column in Yellow

KIT33812ECUEVME Reference Design						**Target Engine System**		
Connector	Pin	Signal Name	Signal Type	Voltage Range	Recommended Functionality	Connector Pin	Wire Color	Functional Description
	1	VPWR	Power Input	13.6V	System power from 12V battery			
	2	ISO9141	Input / Output	0-Vbat	Bi-directional communication pin for diagnostics			
	3	COIL		0-Vbat	Spark control of digital ignition system			
	4	GND		0V	Module level ground reference, return path of Vbat			
	5	GND		0V	Module level ground reference, return path of Vbat			
	6	TPMD		0-Vbat	H-bridge control for 4-phase stepper motor for idle speed air speed control			

Exercise 1: Complete the Load Worksheet for your target engine system.

1. Open “Load Worksheet.xls” and bring the “Instructions” sheet to the front by clicking on this tab.
2. Collect information such as wiring diagrams and schematics for the engine system to be run.
3. Use the engine system information to define how each signal of the ECU is going to be connected to the engine. This includes a definition of an existing pin on a connector, wire color and type, and the functionality associated with the system. This table will also be useful for configuring the software.
4. Repeat this exercise for creating a simulated engine environment.

Creation of simulated engine environment

Before the simulated environment can be created, the ECU must have a viable power source. As the ECU is designed to work in a real engine system, it is required to have a 12 V power source. A power supply capable of generating 12 V at 1.0 A is a good starting point for the ECU alone. Depending on the loads that will be connected to the ECU, a much larger power supply may be required with high current. A good starting point for working with a full featured system is a 12 V, 10 A power supply. While the total system loads may be greater, 10 A is generally large enough since the high current loads of ignition and injectors are not typically on simultaneously.

The best and safest way to begin developing an application for engine control is to work with a simulated engine system. This reduces risk and development time by not having to focus on fuel related safety concerns when trying to solve complex applications issues that arise. Developing with a simulated engine system engine begins by selecting components that are similar or identical to the actual components on the engine. For many of the loads, these can be the exact same components. In some cases, loads can be replaced by a lesser expensive relay or a light. Relays work well for high power loads with dynamic operating frequencies such as ignition and injectors. In those cases, the sound of the relay actuation is beneficial to validate behavior during low speed testing. Other loads work better with lights or LEDs. These are more simple loads that are simply controlled as on or off for long periods of time. Some loads will require the actual load to test, such as an idle speed motor.

Perhaps the most challenging part of the system to simulate is engine position. Two core technologies are used to sense engine position: variable reluctance sensors (VRS) and Hall Effect sensors. The majority of production engines use a VRS for engine position. The advantage with the VRS is cost, while a Hall Effect sensor provides a cleaner output signal. Both types are supported on the ECU. The default configuration is for VRS. Use the schematic to identify the components to remove and populate for using a Hall Effect sensor.

With respect to creating a simulated engine environment, engine position is the fundamental element. Simulating the rotation of the engine can be done in two ways, virtual and physical simulation. A virtual simulation involves a digital re-creation of the spinning crankshaft signal. This is best done by reproducing a Hall Effect Sensor type of output, but there are options for a VRS.   Using a different ECU, such as a basic development board for a Freescale MCU, software can be written to create a the missing tooth output pattern that is produced by a rotating engine using a Hall Effect Sensor. Such programs have already been written for varying degrees of Freescale MCUs. The TOOTHGEN function is a part of a library of functions for the MPC55xx products that have the eTPU peripheral.(ref1) Using a development tool for such a product can allow the creation of a simulated engine position signal. For a VRS, options for a virtual simulation include a combination of PC software with simple custom hardware. Do it yourself (DIY) web sites, such as those for the Mega Squirt products, provide detailed instructions for building your own circuit and provide PC software that can control the generation of the VRS signal based on a simulated signal. (ref2)

While the concept of a virtual simulated engine position signal is very attractive, it lacks fundamental characteristics that come with actual crankshaft of an engine. Since a virtual signal is typically generated by a digital computer, it usually does not account for the real world imperfections of an engine. Specifically, the timing pulses produced by a virtual signal are perfect. While this is a good on paper or visually on a screen, the imperfections in the motion due to production tolerance and jerk associated with cylinder compression lead to a rotation pattern that is not perfect. As a step in the right direction, a physical simulated engine position signal can be used. This type of setup can take advantage of VRS or Hall Effect Sensors and produce a signal that has more realistic characteristics to a real engine. A simple and effective way to make a physical simulation is to mount an engine flywheel containing the position teeth to a small electric motor. This creates a tool known as a spin bench. Using an electric motor and the actual flywheel allows simple control of the engine speed while adding real world conditions for changes in the actual time between position teeth. While the strong variations related to compression and combustion are not present, the spin bench does allow transitions to and from a stopped engine and provide teeth that are representative of the actual engine that the application is being developed for. **Figure 8** shows an example of a spin bench using a production flywheel and VRS from a small motorbike.

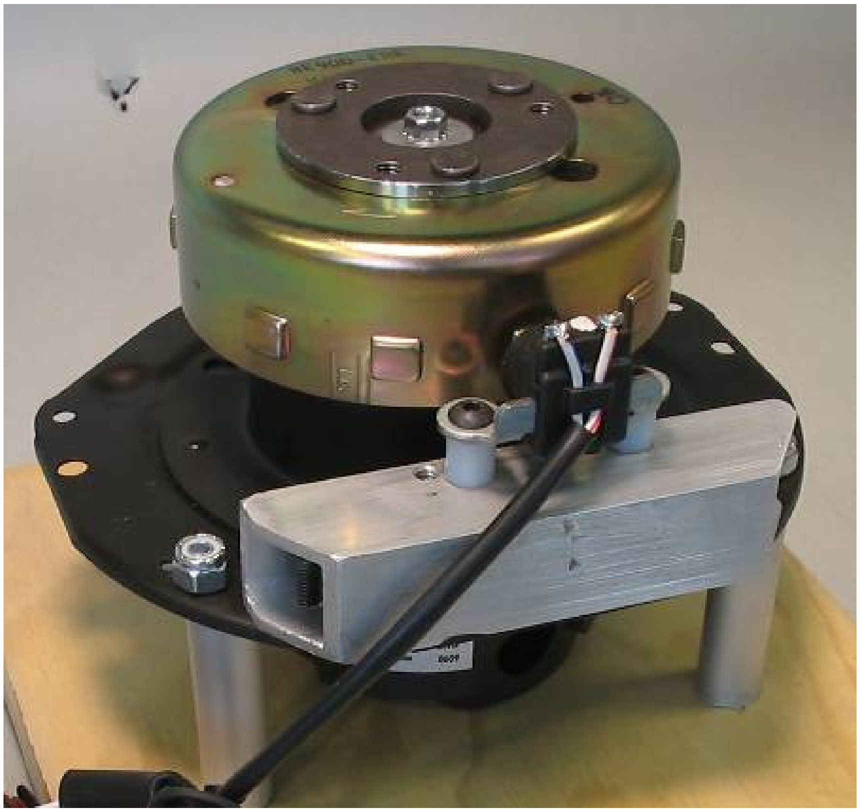


Figure 8. Spin Bench Example for Creating a Physical Simulated Engine Position Signal

Exercise 2: Creating a reference platform for a simulated engine environment

1. Open Load Worksheet.xls and bring the “Reference System Load Worksheet” to the front.
2. Obtain components listed. Generic component specifications are listed.
3. Additionally, a simulation for engine position will be required. Create this using any of the examples described in this section. Verify the simulated engine position signal is being properly generated. For this reference platform to work, a 12 minus 1 signal must be generated. This means 12 equally spaced teeth with one missing tooth representing a gap. See Figure 9 for oscilloscope trace of 12 minus 1 signal.
4. Create a wire harness to connect reference components to the ECU. Include specifications of wire color and pin number as applicable. This will aid in debugging and later development.
5. Connect the wire harness to the ECU.
6. Place the Engine Stop Switch in the active position, which is a short to ground.
7. Apply power. Verify connections are correct by noting that power supply is drawing less than 500mA of current and no components of the ECU are generating large amounts of heat. If any component is hot, remove power and verify connections.
8. Verify that no relays should be active. This should be audible when power is applied if a relay was activated. If relay activates on power on, verify Engine Stop Switch position and relay connections.

9. Move Engine Stop Switch to passive (12 V) position. This should activate the ROUT1 relay for 3 seconds then deactivate the relay. Audibly this will be heard by two clicks. If connections are good and relay is not actuating, verify signal on P1-9 (ISO9141) is low (0 V). If this is not low, then ECU does not have application code and will require programming.

10. Start engine position simulation through Hall Effect or VRS. Keep RPM to about 500RPM. Relay connected to COIL should be turning on and off each rotation and be audibly heard. This indicates that a good engine position signal is getting to the MCU and it is able to process and control the loads.

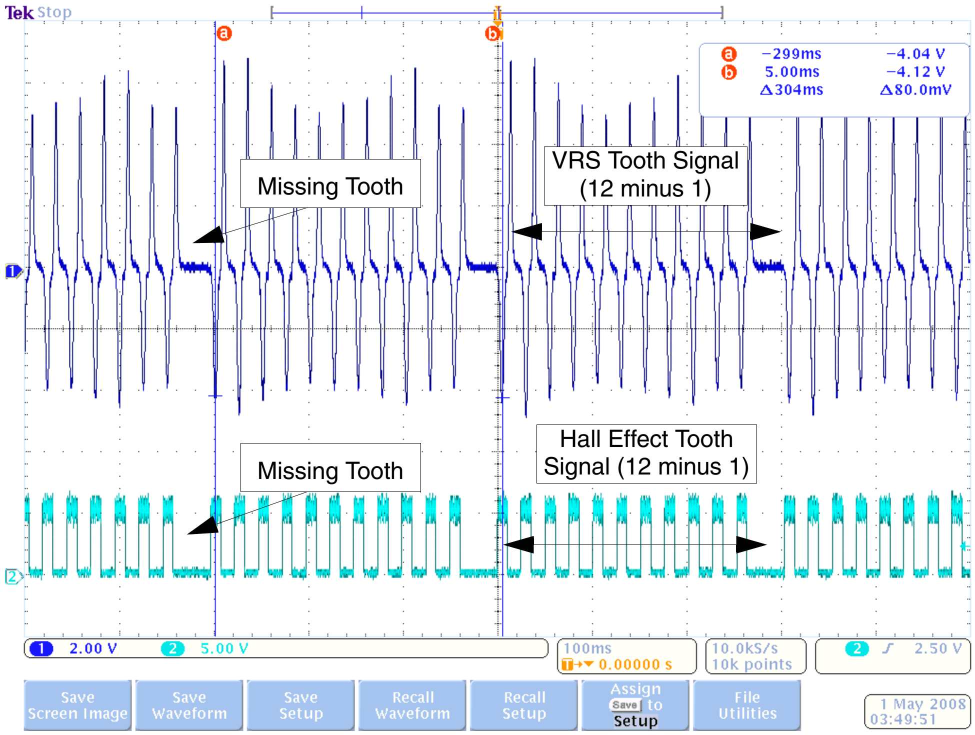


Figure 9. Graphical Representation of 12 Minus 1 Tooth Pattern on Oscilloscope

Installation and Verification of Software Development Environment

All application software for the ECU is developed using the Freescale CodeWarrior for S12(X) integrated development environment (IDE). Version 5.0 is the latest at creation of this manual and is included on the DVD included with the KIT33812ECUEVME. To install CodeWarrior, save then launch the installation application from a temporary location on a Windows based PC or directly launch the installation application. No specific instructions are recommended beyond the default settings shown in the on screen menus. If other versions of the CodeWarrior product are on the PC, this will not overwrite any information as each version is a separate product and installation. For step by step confirmation of the installation process and a quick tutorial on getting started, refer to the CodeWarrior Quickstart Guide included on the Documentation DVD/CD. Further information relative to CodeWarrior can be found at www.freescale.com/training. This link has a search feature allowing refinement of high level training topics. Two training topics that will aid in the use of this kit are learning C programming (http://www.freescale.com/webapp/sps/site/training_information.jsp?code=TP_C_PROGRAMMING&fsrch=1) and the Virtual lab for the S12XE (http://www.freescale.com/webapp/sps/site/virtual_lab_information.jsp?code=VLAB_EVB9S12XEP100&fsrch=1). Additional resources for working with CodeWarrior include the various user manuals that are installed with CodeWarrior.

Once you have installed the CodeWarrior application and become familiar with its operation through the virtual lab, the software development environment can be validated.

Software Development Environment Validation Exercise

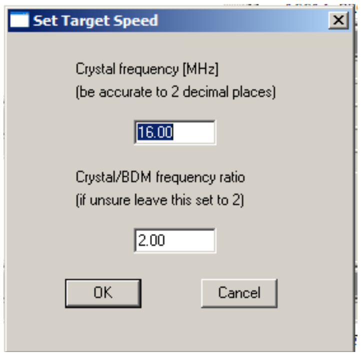
1. Create a new project in CodeWarrior for the MC9S12P128 MCU using the Project Wizard that appears when CodeWarrior is launched. Create the project using default settings but be sure to include the USB BDM tool included with your kit as the target connection.
2. Once you have the project created, verify the integrity of the empty software project by doing a build.
3. Once successful, connect the ECU to your 12 V power source using your simulated load harness.
4. Next, connect the PC to the USB BDM tool. Installation will be required if this is the first connection to the PC, follow on screen menu and install driver automatically.
5. Connect the BDM ribbon cable to the BDM header on the ECU, note the location of Pin 1 as the red wire on the cable and number 1 near the header.
6. Press the debug control in CodeWarrior to download the empty software project to the ECU. Follow the on screen menus to connect and program the ECU, as performed in the Virtual Lab for the S12XE.
7. Press the "GO" arrow and allow execution for a few seconds before pressing "HALT". The source window should show the processor stuck in an infinite FOR loop. This verifies that the ECU is working and the software environment has been created allowing programming and development.

As a final piece towards a complete development environment, a build of the example software will verify if all of the tools discussed this far are working on your system.

System Setup Validation Exercise

1. Save the example application software by copying the folder "Example Scooter Application" from the DVD. This is a CodeWarrior project that contains a working application that runs a 50cc scooter engine
2. Open "My _Engine_Project.mcp" in the saved folder through CodeWarrior.
3. Build the project.
4. Program the ECU by providing power and clicking debugger per previous exercise.
5. Run the application using the green "GO" button.
6. Stimulate the application by running the engine position simulation and using the Engine Stop Switch. Operation should be identical to simulated engine environment test performed above.

Note: When using the TBDML as a BDM tool, the BDM communication speed must be manually changed when the MCU switches between internal and external oscillator settings. The example application switches from internal to external oscillator and it is necessary to change the BDM speed to 8MHz as shown in **Figure 10**. This setting is found in the TBDML HCS12 drop down menu in the debugger window. If you do not have this drop down menu, you do not have the proper connection selected in Codewarrior.



Set Target Speed

Crystal frequency [MHz]
(be accurate to 2 decimal places)

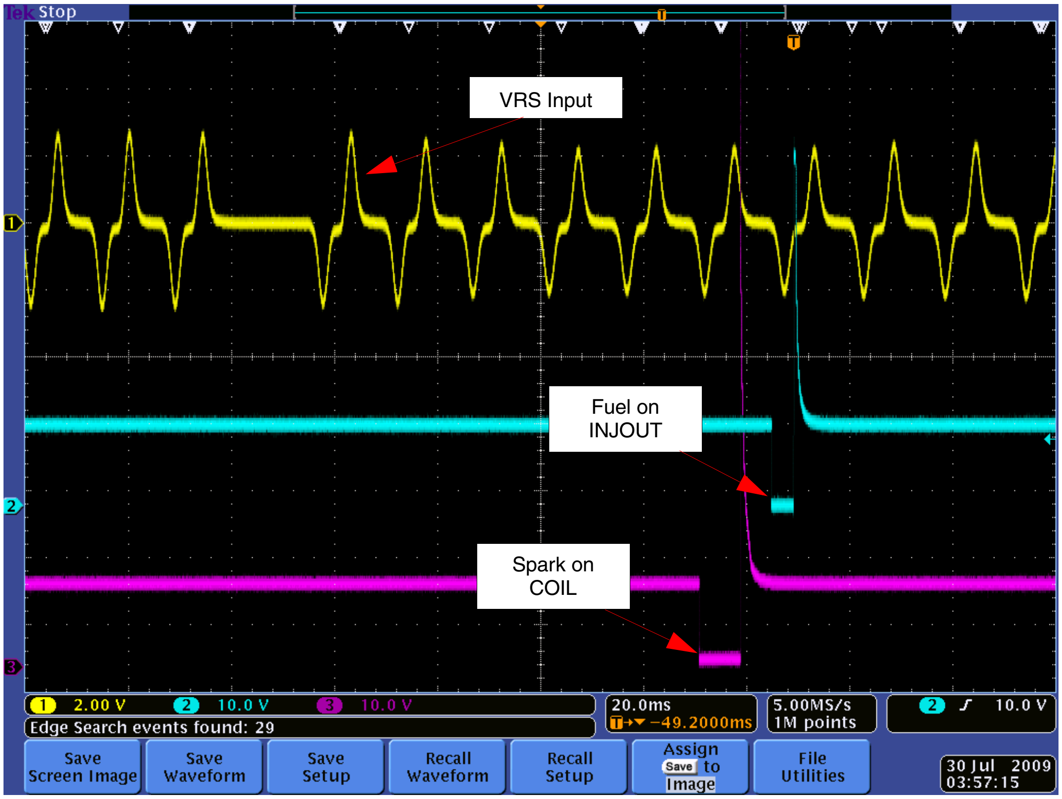
16.00

Crystal/BDM frequency ratio
(if unsure leave this set to 2)

2.00

Figure 10. Changing the BDM Communication Setting for TBDML

7. Verify control signals for VRS, COIL, and INJOUT match those shown in **Figure 11** using an oscilloscope.



Edge Search events found: 29

Figure 11. Control Signals for Reference System Validation Exercise

Congratulations! This is a significant step towards creating your own engine controller. A safe and effective development environment has been created allowing you to create your own application for small engines. As the next sections progress, the focus will be mainly on the C-language source code used in the example application. It is recommended to be experienced in the C programming language to continue.

4 Application Development

There are three paths that can be taken using the Small Engine Reference Design for application development: 1) Ground up custom code can be written. 2) The example applications can be modified. 3) A ground up application can be written using the low level drivers and operating system used in the example applications. If a ground up software project is selected, it may be beneficial to use various aspects of the example application for working with the S12 MCU and the other various components in the design. The example application will also be a benefit when using the low level drivers as it serves as an example for using these pieces of code. At the very least, customizing of the example application will be required. This section will focus on customizing the example application to a specific engine.

4.1 Example Application Architecture Overview

The example application is designed to run a one or two cylinder engine using a hybrid operating system. A hybrid operating system is important to engine control as all engine control events are based on the rotation (angle domain) of the engine and user control processing and data collection must be performed periodically (time domain). Additionally, the example application reduces complexity through a hardware abstraction layer (HAL). Through the HAL, software complexity is reduced by using application level signal names instead of native control names for the MCU. The combination of these two software techniques produces an example that is configurable through a single header file and reduces user implemented code to three main functions.

User functions are split into three main activities. In Data_Management(), all data is collected and processed in the system. This includes analog and digital information and any filter functions that are to be performed. Engine_Management() is called to calculate raw fuel and spark parameters for running the engine. This includes table look up of hard data values based on current engine RPM and load as well as factoring in fuel and spark modifiers. In User_Management(), the engine control strategy is run. It includes interpretations of user control inputs and control strategies for loads. The primary goal of the User Management function is to handle user controls, determine fuel modifiers, and calculate engine load. Each of these functions are performed at various rates and configured through the Application Definitions.h header file. These functions do not directly control the engine fuel and spark events. These are performed by low level functions that react to the rotation of the engine through the engine position data. The low level engine control events use the latest parameters passed to fuel and spark controllers by the user functions. Additional information is provided in Software Reference Manual found in Appendix B.

4.2 Configuring the Application

The first step in working with the example software is to configure the code to be generated through the Application Definitions.h file. In this file are definitions used to conditionally compile code based on the user defined system. This is done to create an application that only uses the memory required for the specific application, demonstrate flexible software design through conditional compiling, and create a framework for a custom implementation using various types of hardware. The file is designed to be simple and allow decisions to which definitions to select by using the completed Load Worksheet, discussed earlier, and knowledge of the application.

While the software provides a signal abstraction layer, configuration of the low level software must be performed through an application header file, “Application Definitions.h”. This file defines what signals are used in system and provides parameters that lead to conditionally compiled code. Example of configuring the software is provided in the demo application. The header file gives you detailed description on how to choose what options you want in your system. Configuring the system through the application header file is done by modifying system parameters by adding or removing specific lines through the comment directive of the C programming language. The following examples goes through various definitions found in the application header file and show possibilities for configuration. It is important to keep in mind the limitations of the hardware as the software incorporates functionality beyond what is found on the reference design hardware.

Example: Configuring the number of cylinders.

```
//How many cylinders? Choose one.
#define ONE_CYLINDER
//#define TWO_CYLINDER
```

To change this application from one cylinder to two cylinders, modify the lines as follows:

```
//How many cylinders? Choose one.
//#define ONE_CYLINDER
#define TWO_CYLINDER
```

Other configuration of the application header file will require modifying parameters that are numerical in nature. Each value must be customized to your application. Default values are provided but may not be relevant.

Example: Configuring maximum RPM of engine.

```
//Set the maximum RPM for engine rotation
#define RPM_MAX 10000
```

This parameter can be modified to reduce the maximum RPM from 10 KRPM to 500 RPM as follows:

```
//Set the maximum RPM for engine rotation
#define RPM_MAX 500
```

For system signals that are configurable, multiple definitions are required. Only if the signal is used do any of the associated parameters need to be defined.

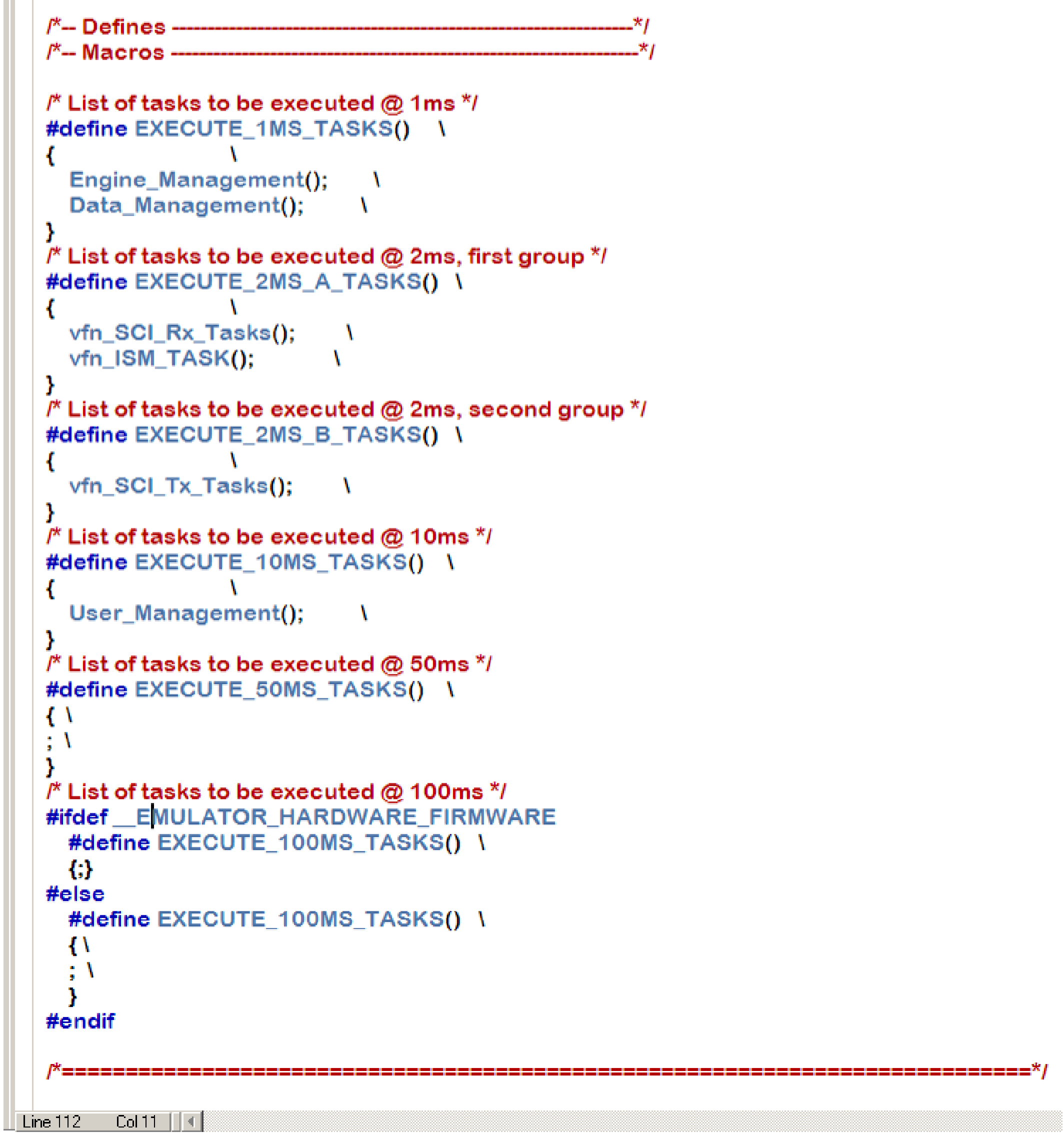
Example: Removing definition of an analog signal.

```
//Oxygen Sensor(O2)
//Define the signal for the system to enable functionality.
#define O2
//Define for O2 filter algorithm selection. Only average is
//available.
//Leave undefined for using raw data only.
#define AVERAGE_FILTER_O2
//Data collection periodic rate can be from 1 - 255ms.
#define O2_DATA_COLECTION_RATE 16
/* O2 data buffer size */
 #define O2_BUFFER_SIZE 16
```

In this example, if the Oxygen Sensor is not used, then all pound defines should be changed to comments as follows:

```
//Oxygen Sensor(O2)
//Define the signal for the system to enable functionality.
//#define O2
//Define for O2 filter algorithm selection. Only average is
//available.
//Leave undefined for using raw data only.
//#define AVERAGE_FILTER_O2
//Data collection periodic rate can be from 1 - 255ms.
//#define O2_DATA_COLECTION_RATE 16
/* O2 data buffer size */
//#define O2_BUFFER_SIZE 16
```

One additional configuration is provided outside the Application Definitions.h file. This is the configuration of the time domain scheduler of the operating system. Configuration of the timing for the tasks is done in the Tasks.h file. As seen in **Figure 12**, the various tasks are configured by placing function calls in the desired task time. While this is an easy way to implement a variety of time based tasks, this simple scheduler does not guarantee task execution time. It is recommended to perform timing analysis using simulation and instrumented software as a part of the application development process.



```
/*-- Defines ------------------------------------------------------------*/
/*-- Macros -------------------------------------------------------------*/

/* List of tasks to be executed @ 1ms */
#define EXECUTE_1MS_TASKS()    \
{                        \
   Engine_Management();      \
   Data_Management();       \
}
/* List of tasks to be executed @ 2ms, first group */
#define EXECUTE_2MS_A_TASKS()  \
{                        \
   vfn_SCI_Rx_Tasks();       \
   vfn_ISM_TASK();          \
}
/* List of tasks to be executed @ 2ms, second group */
#define EXECUTE_2MS_B_TASKS()  \
{                        \
   vfn_SCI_Tx_Tasks();       \
}
/* List of tasks to be executed @ 10ms */
#define EXECUTE_10MS_TASKS()   \
{                        \
   User_Management();       \
}
/* List of tasks to be executed @ 50ms */
#define EXECUTE_50MS_TASKS()   \
{ \
; \
}
/* List of tasks to be executed @ 100ms */
#ifdef __EMULATOR_HARDWARE_FIRMWARE
   #define EXECUTE_100MS_TASKS()  \
   {;}
#else
   #define EXECUTE_100MS_TASKS()  \
   { \
   ; \
   }
#endif

/*===========================================================================*/
```

Figure 12. Definition of Tasks in Tasks.h File

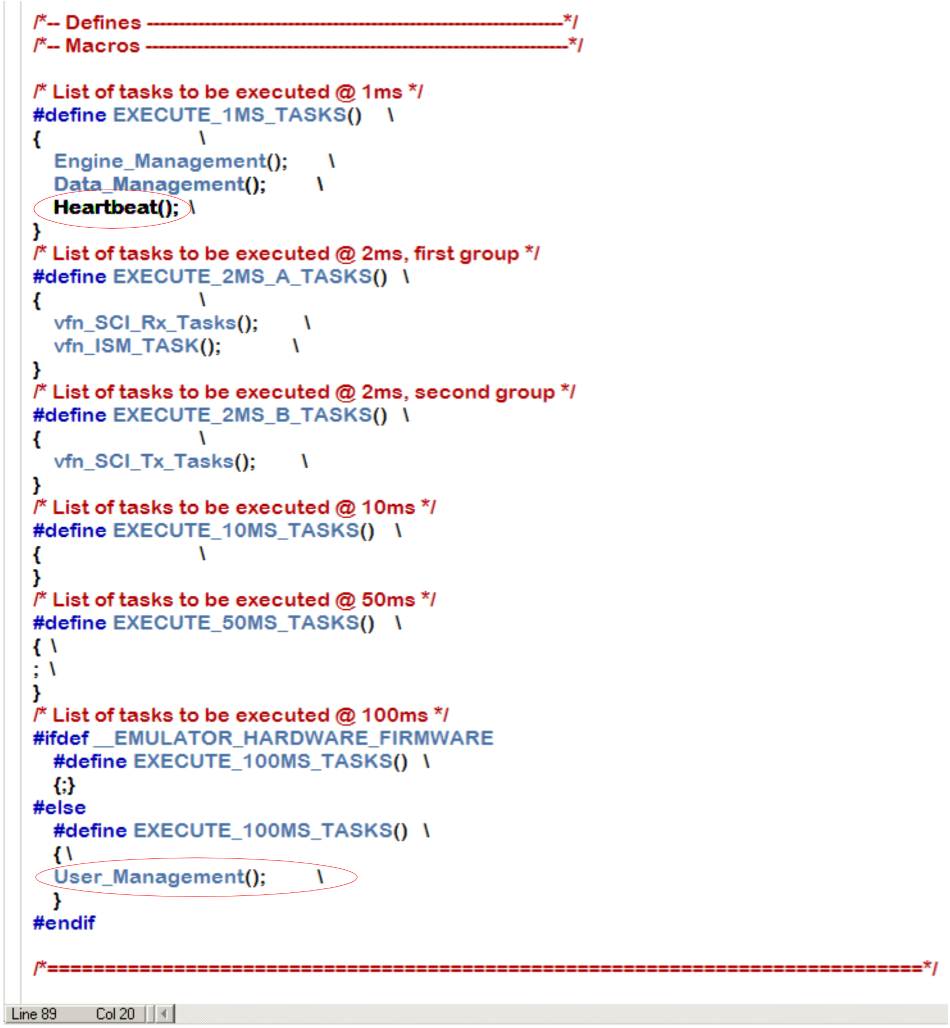
To configure the task timing, edit the definitions shown in **Figure 12** using the exact syntax found in the file.

Example: Modifying Task Times

In this example, the default task scheduler found in the example application will be modified to show how to slow down the execution of User_Management() and add a custom function to be run every 1ms called Heartbeat().

1. Open the example application using CodeWarrior.
2. Open the file “Tasks.h”.
3. Find the definition section containing the 10 ms tasks.
4. Select the line containing the function call “User_Management()”. Cut this line from the code.
5. Place the User Management task by copying it into the space for 100 ms tasks.
6. In the 1.0 ms task section add a line containing the function call “Heartbeat()” and follow syntax shown for other tasks

When complete the code shown in **Figure 12** should look exactly like the code shown in **Figure 13** below.



```
/*-- Defines -----------------------------------------------------------*/
/*-- Macros ------------------------------------------------------------*/

/* List of tasks to be executed @ 1ms */
#define EXECUTE_1MS_TASKS()    \
{                      \
   Engine_Management();     \
   Data_Management();       \
   Heartbeat(); \
}
/* List of tasks to be executed @ 2ms, first group */
#define EXECUTE_2MS_A_TASKS()  \
{                      \
   vfn_SCI_Rx_Tasks();      \
   vfn_ISM_TASK();          \
}
/* List of tasks to be executed @ 2ms, second group */
#define EXECUTE_2MS_B_TASKS()  \
{                      \
   vfn_SCI_Tx_Tasks();      \
}
/* List of tasks to be executed @ 10ms */
#define EXECUTE_10MS_TASKS()   \
{                      \
}
/* List of tasks to be executed @ 50ms */
#define EXECUTE_50MS_TASKS()   \
{ \
; \
}
/* List of tasks to be executed @ 100ms */
#ifdef __EMULATOR_HARDWARE_FIRMWARE
   #define EXECUTE_100MS_TASKS()  \
   {;}
#else
   #define EXECUTE_100MS_TASKS()  \
   { \
   User_Management();       \
   }
#endif

/*===========================================================================*/
```

Figure 13. Modifications of Tasks.h from Example Exercise

4.2.1 Fuel and Spark Data Tables

As a means to input data used for fuel and spark values, an Application Map Tool based on a spreadsheet is provided. This tool provides the essential functionality for translating fuel and spark data tables into content that can be placed into the example software. Specific engine management data can be placed into the tool using engineering units. This table is then converted to microcontroller units in a C-source friendly format. Map table sizes can be adjusted to meet application requirements. The Application Map tool is identified as “Map Tool.xls”. Additionally, reference for an example map is provided in “Scooter Map.xls”. This provides an example of a completed map as used in the example application.

4.2.2 Modifying Table Sizes

As a first step, the table size should be customized to accommodate the performance and data requirements. This is accomplished by adjusting the number of load points and RPM points in the table. In the empty map provided (Map Tool.xls), this is done by changing number of and content of the load row (green) and the RPM column (yellow) values. Both the number of load and RPM values directly impact the size of the table and speed at which the table look up is performed.

While more data points gives you better tuning ability, it will increase the size of the application and increase the worst case time to perform the table look up. Another factor used for sizing the tables is available data. If a legacy map is used then the simplest starting point is to directly reuse this map. If a new map is to be created by empirical data through testing, a smaller map is the best starting point.

Fuel and spark maps are independent of each other and the load and RPM points must be customized for both sets of data. Using the “Fuel Engineering Units (ms)” and the “Spark Engineering Units (BTDC)” worksheets, enter the desired number of points and values for each point for the load row and RPM column. Load is input as a percentage from 0 to 100% in ascending order, left to right. RPM is input from 0 to your max RPM in ascending order, top to bottom.

When determining your max RPM, you should consider the performance of the engine as well as the resolution of the software. For the example application software, a fundamental timing unit is 1.6 μs. This means that the highest resolution between RPM measurements is 1.6us. However, RPM, or engine speed, is determined from the tooth period measurements on the engine's flywheel. This means is that while the engine is rotating at a given RPM, the measurement taken is at a fraction of this rate.

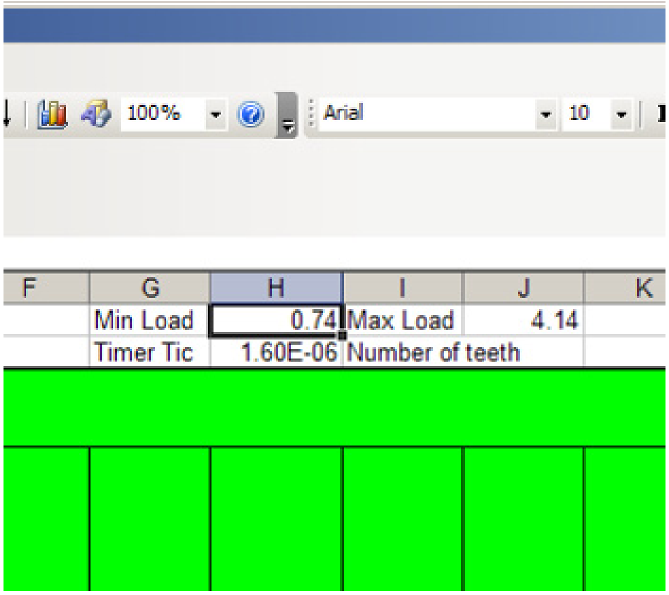
For example, at 6000 RPM, an engine completes one rotation every 10 ms. The engine controller monitors position of the engine through the teeth on the flywheel. Each engine will have a specific number of teeth. For this example the engine has 12 teeth. The result is that the engine controller will measure the time between two teeth at 6000 RPM as 833 μs. Looking at our fundamental timing unit, the software will provide a measurement of 520 (really 520.8 but quantization results in 520).

At 6000 RPM, there is not much sensitivity due to the 1.6 μs timing unit as there is a count of 520. However, as the RPM and number of teeth increases so does the sensitivity. This concept is important to understand and also is relevant for low RPM conditions as well. At low RPM maximum time that can be measured is 104.5 ms. For the 12 tooth engine example, this would correspond to 47 RPM.

4.2.3 Configuring Data Translation

Before entering any data, the parameters used to translate engineering units to MCU units must be properly set. This must be done on two worksheets: “Fuel MCU Units (Tics & Counts)” and “Spark MCU Units (BTDC & Tics).” At the top of these two worksheets are five parameters that each must customized to each engine system.

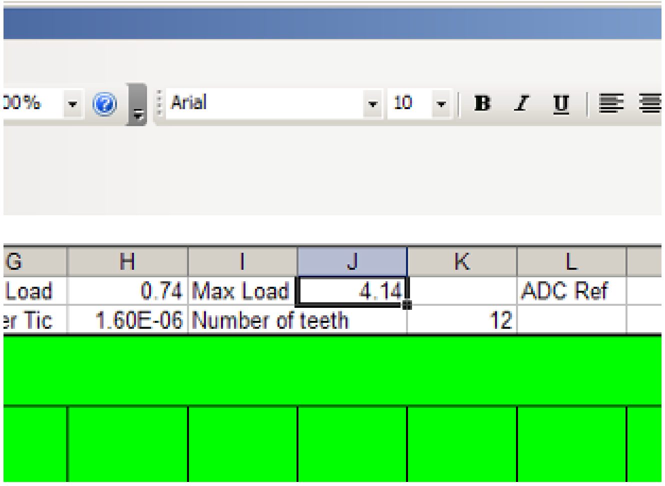
Use the Min. Load value to change what the minimum voltage reading is for load. For a throttle position based load, this is typically the closed throttle position. For a MAP sensor based load, this is the voltage produced by the MAP sensor at a minimum operating pressure.



F	G	H	I	J	K
	Min Load	0.74	Max Load	4.14	
	Timer Tic	1.60E-06	Number of teeth		

Figure 14.

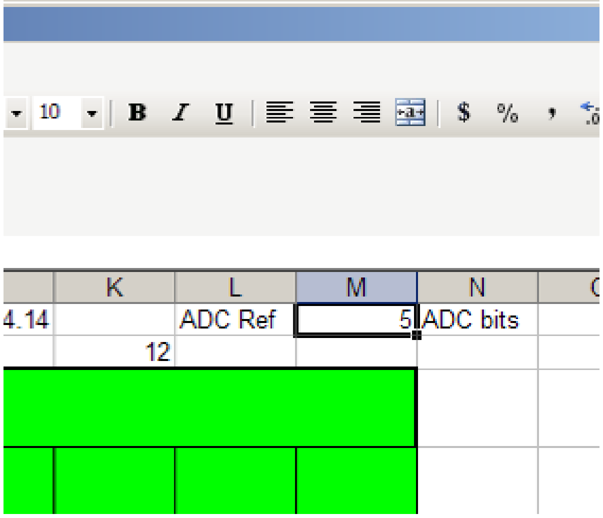
In the Max Load field, change the value to what the maximum voltage reading is for load. The same concepts apply as for Min. Load only this is at a minimum condition.



G	H	I	J	K	L
Load	0.74	Max Load	4.14		ADC Ref
er Tic	1.60E-06	Number of teeth		12	

Figure 15.

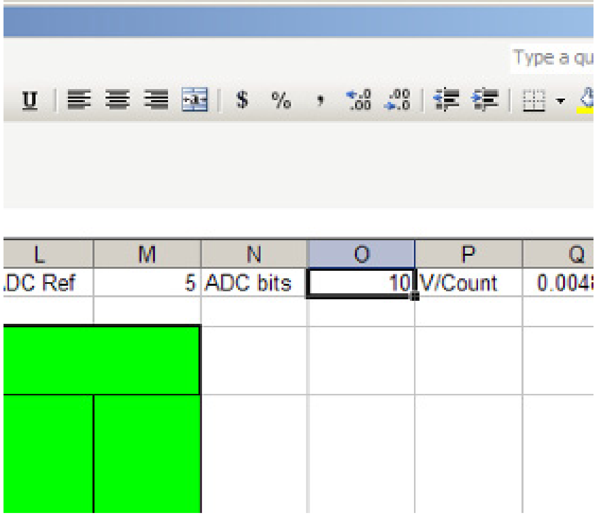
For the ADC ref field, input what the reference voltage is for the analog measurement. The Small Engine Reference Design uses 5.0 V as the reference and this should not be changed.



	K	L	M	N
4.14		ADC Ref	5	ADC bits
	12			

Figure 16.

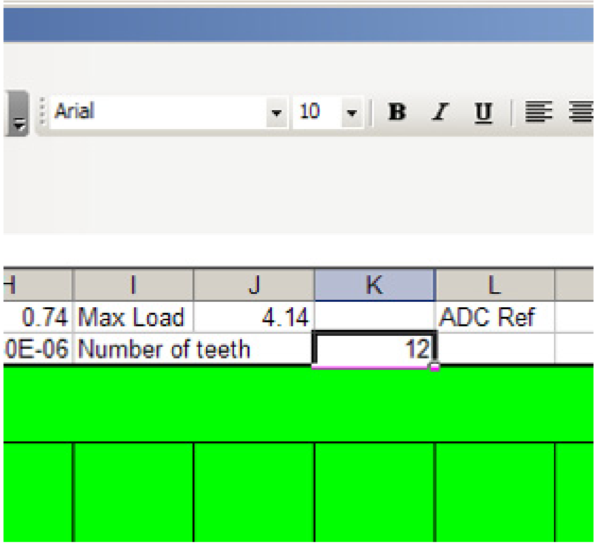
Depending on the software configuration, analog data is collected as 8, 10, or 12 bits. Make sure this field matches how the software is configured.



L	M	N	O	P	Q
DC Ref	5	ADC bits	10	V/Count	0.004

Figure 17.

The final field that must be completed is the Number of teeth. This is the number of teeth on the flywheel as used for synchronization and engine speed measurement. Use the total number of teeth including missing teeth as the spacing is the important characteristic. For example, an engine may have a 12 minus one tooth configuration, meaning 12 equally spaced teeth and one of the teeth is removed for synchronization. In this case the relevant number is 12.

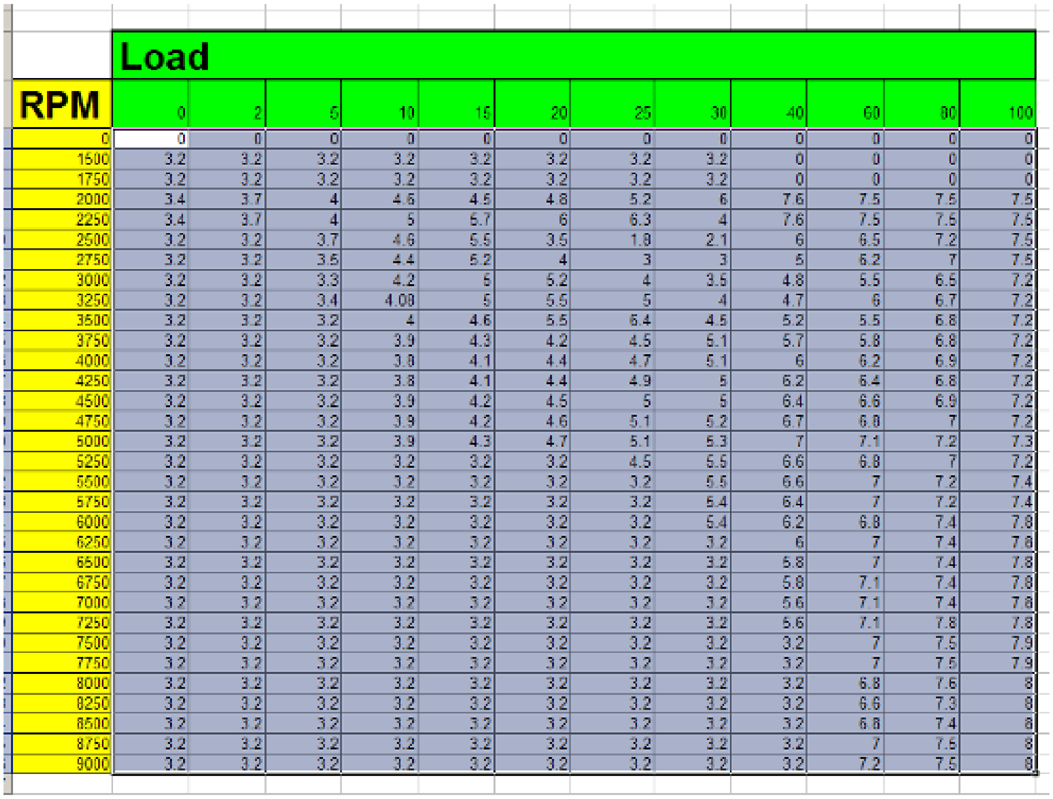


H	I	J	K	L
0.74	Max Load	4.14		ADC Ref
0E-06	Number of teeth		12	

Figure 18.

4.2.4 Entering Map Data

Each combination of load and RPM value creates a unique data point that can be accessed by the software during execution. The data for fuel and spark maps are input into the “Fuel Engineering Units (ms)” and the “Spark Engineering Units (BTDC)” input worksheets, respectively. As data is entered in these two worksheets, it is translated on the “Fuel MCU Units (Tics & Counts)” and the “Spark MCU Units (BTDC & Tics)” output worksheets. These two output worksheets contain the same data as the input worksheets only translated based on the MCU and software configuration.



	Load											
RPM	0	2	5	10	15	20	25	30	40	60	80	100
0	0	0	0	0	0	0	0	0	0	0	0	0
1500	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	0	0	0	0
1750	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	0	0	0	0
2000	3.4	3.7	4	4.6	4.5	4.8	5.2	6	7.6	7.5	7.5	7.5
2250	3.4	3.7	4	5	5.7	6	6.3	4	7.6	7.5	7.5	7.5
2500	3.2	3.2	3.7	4.6	5.5	3.5	1.8	2.1	6	6.5	7.2	7.5
2750	3.2	3.2	3.5	4.4	5.2	4	3	3	5	6.2	7	7.5
3000	3.2	3.2	3.3	4.2	5	5.2	4	3.5	4.8	5.5	6.5	7.2
3250	3.2	3.2	3.4	4.08	5	5.5	5	4	4.7	6	6.7	7.2
3500	3.2	3.2	3.2	4	4.6	5.5	6.4	4.5	5.2	5.5	6.8	7.2
3750	3.2	3.2	3.2	3.9	4.3	4.2	4.5	5.1	5.7	5.8	6.8	7.2
4000	3.2	3.2	3.2	3.8	4.1	4.4	4.7	5.1	6	6.2	6.9	7.2
4250	3.2	3.2	3.2	3.8	4.1	4.4	4.9	5	6.2	6.4	6.8	7.2
4500	3.2	3.2	3.2	3.9	4.2	4.5	5	5	6.4	6.6	6.9	7.2
4750	3.2	3.2	3.2	3.9	4.2	4.6	5.1	5.2	6.7	6.8	7	7.2
5000	3.2	3.2	3.2	3.9	4.3	4.7	5.1	5.3	7	7.1	7.2	7.3
5250	3.2	3.2	3.2	3.2	3.2	3.2	4.5	5.5	6.6	6.8	7	7.2
5500	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.5	6.6	7	7.2	7.4
5750	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.4	6.4	7	7.2	7.4
6000	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.4	6.2	6.8	7.4	7.8
6250	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	6	7	7.4	7.8
6500	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.8	7	7.4	7.8
6750	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.8	7.1	7.4	7.8
7000	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.6	7.1	7.4	7.8
7250	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	5.6	7.1	7.8	7.8
7500	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	7	7.5	7.9
7750	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	7	7.5	7.9
8000	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	6.8	7.6	8
8250	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	6.6	7.3	8
8500	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	6.8	7.4	8
8750	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	7	7.5	8
9000	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	7.2	7.5	8

Figure 19. Completed Input Table for Fuel Map Data Example

An example of using this tool is provided and is the actual maps used by the demo software running a scooter engine. This serves as a reference and should not be considered a starting point for any engine without validation. Validated maps from other fuel management systems can be directly input into this tool if in the same format.

4.2.5 Exporting Map Data

Once the fuel and spark maps are completely filled, it is necessary to export the data to a file format that is C-source code friendly and can be placed into the example application. This is accomplished by saving the worksheets labeled as “Fuel Export Data” and “Spark Export Data” in a comma delimited format and performing limited modification to the saved file. Once the data is then saved in this new format, it can be copied and pasted into the Sea Breeze Emulator Software.

4.2.6 Map Data Export Process

1. Complete fuel and spark map data entry per above desciption.
2. Select the “Fuel Map Export Data” as the active worksheet.
3. Verify the table values match with the values of “Fuel MCU Units (Tics & Counts)”.

4. With “Fuel Map Export Data” active, save the file as a comma delimited file with the extension <my fuel map>.csv. This will put the active worksheet into a file that contains only the fuel data separated by commas. When saving this file as a “.csv”, many warnings will be presented. Most of these warnings will indicate that the new file format does not support multiple worksheets. Read these warnings and select the option that saves the active worksheet and continues with the operation.

5. The “.csv” file will need one specific alteration. Open the fuel map <my fuel map>.csv using a text based editor, such as WordPad. At the end of each row of data, add a comma after the last data value, excluding the last row. Save the file. The data can now be copied and pasted into the Application Map.c file of the Sea Breeze Emulator Software in the fuel data array. Choose the array that fits your data type as configured in the map tool and the application header file.

6. Repeat steps 1 through 5 for spark map data.

Additionally, information regarding the size of the table and the actual values of for each of the load and RPM values must be put into the Application Map.c file. The same process used for the table data can be used for the load and RPM values using specific export tabs and above procedure provided. The number of load and RPM points for the fuel and spark arrays must be put into the Application Map.h file. It is up to the user to ensure the table is sized properly for the data that is input into the actual map. Errors in the size of the data tables or the data used for each load or RPM value will result in an improper table look up procedure, which may result in random data used to create fuel and spark events. Use the demo application as a guide if there is doubt in your procedure.

4.2.7 Working with the Example Application

The demo application is based on a simple application state machine (ASM) for engine control. This state machine executes in the User_Management() task and can be found in the User_Management.c file. A combination of user controls and engine operating parameters are used to control the states of the application. The five states of the ASM are: INIT, STOP, START, RUN, and OVERRUN. A function call is provided for transitioning to each state. This allows a more controlled engine operating mode when changing states.

Description of User Management States

INIT

This state provides a known configuration of the User Management task and should be configured as the initial state using User_Management_Init(). Variables for User Management should be initialized and any essential activity that is necessary to be performed prior to operating in any other state should be done in the INIT state. Once this activity completes, the ASM should transition to the STOP state where the periodic activity begins. Optionally, if a major system error occurred, the user may find it necessary to return to this state.

STOP

In this state, the engine has been decided to be stopped from rotating or running. System inputs such as switches would typically cause the application to enter the STOP state. The application should configure any outputs or controllers to match this request to stop the engine and remain in this state until the inputs reflect going to an active engine state.

START

As provision for a slowly rotating engine or in preparation for the engine to begin rotating, the START state allows the application to initialize engine controls for an active mode. This state is maintained as long as the engine stays below a minimum speed, identified in the User Management header file as the stall speed. Additionally, the same system inputs that allowed the exit of the STOP state must be present or a transition to the STOP state would occur.

RUN

Once a minimum engine speed has been obtained and the correct system inputs have been applied, the RUN state represents the normal operating state of the application for a rotating or running engine. The engine control strategy is to be implemented in this state. System inputs must be maintained to keep the engine in the RUN state and the engine speed must be above the stall speed but below the maximum speed, identified in the User’s Management header file as over speed.

OVERRUN

As a special case for an active engine, the OVERRUN state provides a way to limit the engine speed. This can be implemented by changing the engine control outputs through variables or through disabling specific engine control outputs. System inputs for an active engine state must be maintained to prevent the ASM from going to the STOP state.

Additionally the engine speed must be reduced below a specific value. This parameter is adjusted in the User Management header file as over speed recovery.

The true performance of the Small Engine Reference Design can only be shown in a real application. Through development using a real engine, testing can be performed that addresses real system issues with an engine control application. Using a real production scooter as a test platform demonstrates the capabilities of the hardware and software beyond documentation. For this purpose, a demonstration application using the 50cc EFI motorbike was selected. By retrofitting the engine controller with the Small Engine Reference Design, a basic engine management application is demonstrated.

5 System Overview

Design for the ECU is centered on an EFI system capable of closed-loop control for a one cylinder engine using known loads. This system includes essential inputs and outputs for an application that is emissions sensitive. Figure 20 shows the high level signals of the Small Engine Reference Design. Inputs include 6 analog channels and one digital input. The names of these signals are suggestions for use but may be changed based on application demands. Additionally, a special input is used for the crankshaft sensor. This is for a VRS using 2 pins but can be adapted to use a Hall Effect type of interface. A total of 10 outputs are available, each with specific functionality in mind. In most cases these outputs are oversized. This allows the drivers to be flexible to a broad range of loads. One special output is provided as a system reference voltage available for sensors drawing small amounts of current. Further detail of ECU design is covered in the Hardware Reference Manual found in Appendix A.

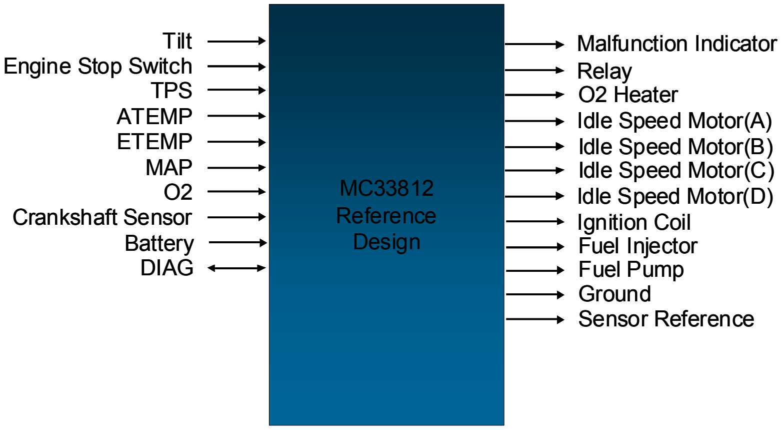


Figure 20. Module Level Signals of the Small Engine Reference Design.

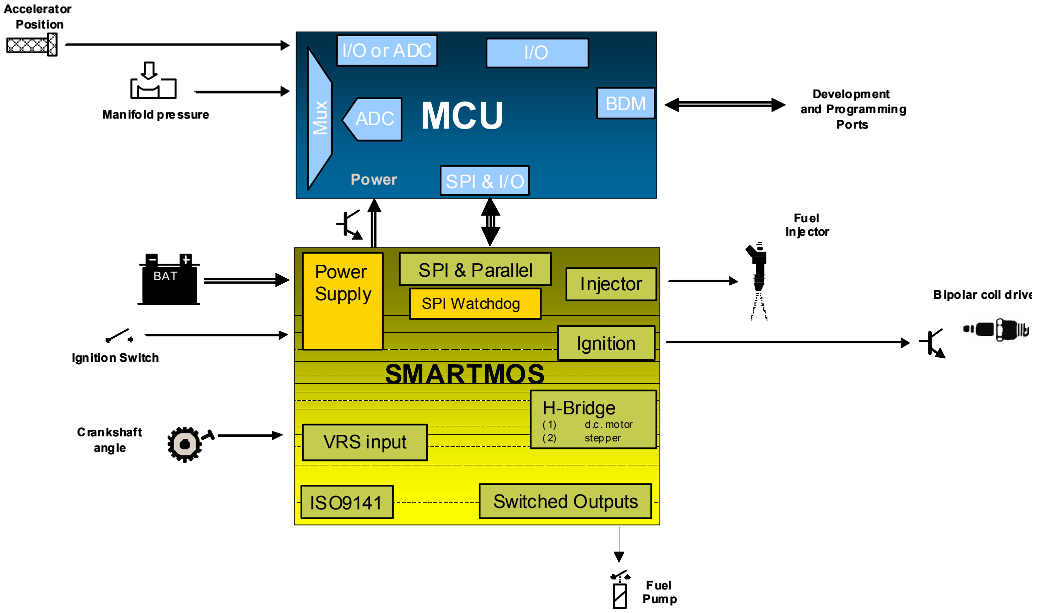


Figure 21. System Block Diagram for the Scooter Engine Control

6 Application Overview

The scooter demo application uses the same simple state machine that described above. Complexity of the application is limited to an Alpha N engine management strategy with system modifier parameters. Engine management strategy uses table look up for fuel and spark parameters based on throttle position and engine speed. Although table look up is restricted to exact values on the table, interpolation between points is can be easily implemented through custom code. Once the table look up value has been obtained for fuel and spark, various modifier values can used to obtain the final value used by the fuel and spark controllers.

These fuel and spark modifiers are determined and managed by the User Management task. The example application only uses two modifiers in the application but provides the framework for an advanced algorithm. These variables are maintained in the User Management task but are added to the table look up base value in a low level function. This action is performed in u16Calc_Fuel_Pulse_Width(), which is used by the Engine Management task. Additionally, two fuel variables are available for tuning that let you directly control the fuel pulse output when a fuel event is scheduled. These two variables are Fuel_Cut and Fuel_Add and have companion variables for spark, referenced as Spark_Advance and Spark_Retard. These direct fuel and spark modifiers are extremely useful for coarse and fine tuning of the engine without the complexity of multiple modifiers.

For starting conditions, a fuel modifier value, FM_MSTART, adds additional fuel to the base look up table value. When in the START state, a calibration value of MSTART is used. When the engine speed increases to the RUN state, the FM_MSTART fuel modifier is decayed at a specific rate, MSTART_Decay_Timeout, based on execution of the User Management task. The amount of the decay is set by MSTART_Decay. Both values are sent in the User Management header file. This working example shows how others can be used in a more advanced implementation.

For transient operation, the fuel modifier FM_MACCEL is used. Parameters associated with the transient fuel detection are found in the user management header file. Tip in and tip out can be detected and individually handled. Thresholds based on ADC count changes between new TPS_Filtered data values can be configured along with decay size and rates. Transient fuel operation is intended to emulate the accelerator pump functionality of a carburetor.

This application also demonstrates active load control through time-outs. The example that is used is the fuel pump. When the application state machine is in the STOP state, the application must be ready to begin starting the engine at anytime. As a strategy, pressurizing the fuel system is best done before any attempt to fuel the engine is made. As a result, the fuel pump is turned on in the STOP state. However, if the fuel pump was left on unconditionally, the battery would be discharged quickly. To prevent this, a time-out is used so that the fuel pump is turned off after 3 seconds in the STOP state. This parameter is defined in the User Management header file as FUEL_PUMP_TIMEOUT and the fuel pump activity is controlled by the Fuel_Pump_Controller(). This simple routine is provided to show how time based management of the loads can be done using engine operating states.

For a four-stroke engine, the process is a 720 degree cycle. Additional synchronization techniques must be performed to transfer the low level processes from a two-stroke to a four-stroke operating mode. For a one cylinder engine, a simple technique can be done to eliminate the need of an additional camshaft sensor. If the system uses manifold absolute pressure (MAP), measurements can be performed to determine if the engine is in a compression or exhaust stroke. This requires looking for a specific signature of the MAP signal. This must be done at the low level. To do this, a toothed based MAP data collection process is selected in the application header file. Specific teeth for collecting data are defined. Signature detection is implemented in the Crank_State_ISR() in the SYNCHRONIZED state in Crank_Sensing.c file. This signature detection algorithm can be easily customized. Implementation using a cam sensor is also easily accommodated for as a simple flag enables the transition from the two-stroke mode to four-stroke mode.

In the application state machine, clear areas are defined to implement engine strategy. A simple example is provided based on using an Alpha N engine control strategy. The software is not intended to show an optimized strategy, however it is designed to provide a starting point for development of an engine management application that is modular and can be successful in running an engine. Thorough understanding and optimization are required when applying this software to any engine application.

Updating Low Level Code

There are no committed support plans for the example application software. It is recommended to continue the customizing of the software down to the lowest level if it is to be used beyond a demonstration purpose. This would include correction of errors for compilations and operation as well as application hardening of software both at the application level and microcontroller level. In event of a significant low level driver update or third party software addition, low level code can be updated in two different ways.

Method 1: Using the new software project containing the new low level driver code, copy in the contents of the user functions. These include User_Management(), Engine_Management(), Data_Management(), and Application Map data. Include any custom definitions in the header files as well.

Method 2: Import new driver files on an individual basis. For specific file updates, copy and overwrite the specific file and header file of interest into the project directory folder structure. Be sure to do a clean build by removing all object code as well as back up the project before overwriting files. Additional dependencies may need to be considered as new code could make use of other new code not found in the replaced file.

6.1 Application Testing

Before going to a real system with your application, extensive testing is recommended to ensure that any engine control signals, specifically fuel and spark, meet your timing requirements and the application provides the desired high level operation and user control. This is best achieved with the simulated system set up previously. Once the application is validated at the bench level, it may be necessary to reduce system functionality and test basic operation when migrating to real engine hardware. Data collection, load control, and system start/stop conditions can be easily tested without the engine running. Additional testing can be performed with fuel and spark controls physically disabled which provides an opportunity for analysis of the rotating engine with no combustion events. Control signals for fuel and spark can then be verified using the actual loads in preparation for calibration of engine control parameters.

Testing the application can be done in real-time using the Hi-wave debugger built into the CodeWarrior IDE. This tool allows non-intrusive debugging through the microcontroller’s BDM pin while the processor is running. Using the Hi wave debugger will allow you to set breakpoints, step though code, view and modify software variables, and directly control the registers of the S12X microcontroller. Assistance learning the debugger can be found in the Virtual Lab mentioned in previous sections. Through the use of calibration variables that are able to be modified using the debugger, testing can be performed to find working values for system control parameters.

One of the most difficult aspects to master is the angle based operation of the engine. It may take many iterations of software building and testing to get the timing right for the delivery of the fuel and spark pulses. Some of this confusion may come from the generic implementation of the example application. The crankshaft state machine that synchronizes to the rotation of the engine, through the teeth pulses on the crankshaft, uses the missing tooth as the definition of top dead center or 0 degrees. The first tooth after the missing tooth gap is then tooth 1. This is not always the case and an offset will need to be created to compensate for specific engine implementation. In the example application, the missing tooth gap is actually at bottom dead center. This creates an offset of 6 teeth (12 minus 1 wheel defined) that must be accounted for. The end result in this case is the top dead center must be defined as tooth 7, creating an offset of 180 degrees.

Once this is compensated for in the application, adjustment of the timing for fuel and spark is handled by the application maps. It may be necessary to create a test map for fuel and spark to verify timing is being properly controlled. Small test maps help provide useful validation tools. Test maps with all the same data are good for basic timing validation. Progressive value maps provide a way to validated table look up is performed as expected.

In the example application, one of the most important calibration variables is the MSTART. This is a fuel modifier variable that adds additional time to the fuel pulse when the application is in the START state. Proper setting of this variable will depend on the engine and fuel injector used, as well as atmospheric conditions. With the debugger open and the application running, it is possible to adjust the MSTART value between starting attempts to adjust the amount of extra fuel is added during startup. This is a delicate process and leads to the art of engine calibration. As calibration is performed, it is important to get back to a steady state before testing a new value. Specifically for MSTART, it is difficult as the failure to start and run the engine can leave the engine cylinder with raw fuel and easily foul the spark plug.

As development of your application progresses, it will need to incorporate features for performance and safety. It is advised to balance hard testing with simulation exercises. This is a necessary step in the development process that should account for worst case code execution. Independent exercises for RAM utilization and execution timing should be considered so that bandwidth of the system is not at 100%, which can easily result in poor engine operation or runaway code due to stack overflow. This leaves room for addition unforeseen complexity that can only come code execution on the real system.

7 Schematic

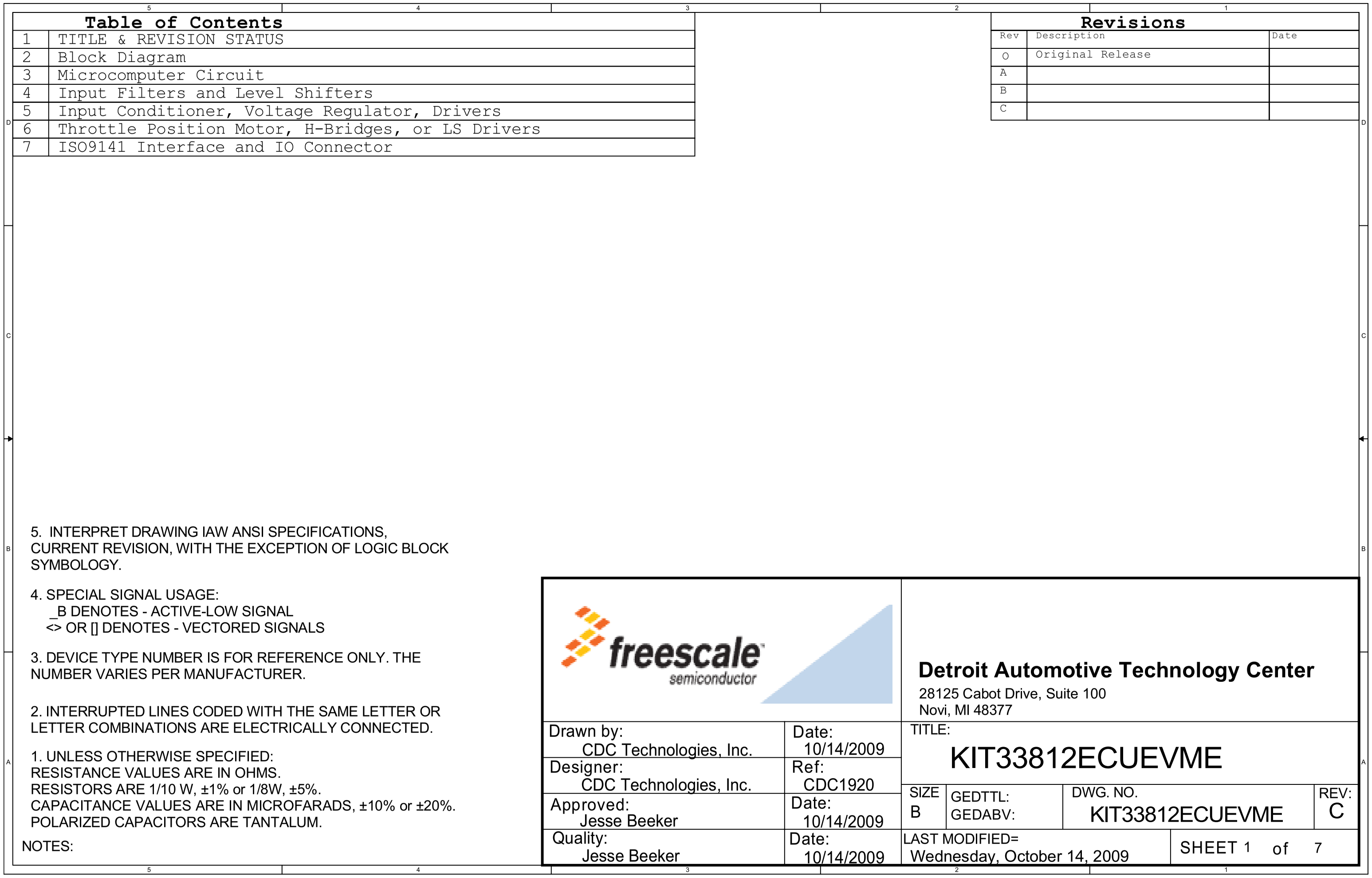


Table of Contents	
1	TITLE & REVISION STATUS
2	Block Diagram
3	Microcomputer Circuit
4	Input Filters and Level Shifters
5	Input Conditioner, Voltage Regulator, Drivers
6	Throttle Position Motor, H-Bridges, or LS Drivers
7	ISO9141 Interface and IO Connector

Revisions

Rev	Description	Date
O	Original Release	
A		
B		
C		

NOTES:

1. UNLESS OTHERWISE SPECIFIED:
RESISTANCE VALUES ARE IN OHMS.
RESISTORS ARE 1/10 W, ±1% or 1/8W, ±5%.
CAPACITANCE VALUES ARE IN MICROFARADS, ±10% or ±20%.
POLARIZED CAPACITORS ARE TANTALUM.

2. INTERRUPTED LINES CODED WITH THE SAME LETTER OR LETTER COMBINATIONS ARE ELECTRICALLY CONNECTED.

3. DEVICE TYPE NUMBER IS FOR REFERENCE ONLY. THE NUMBER VARIES PER MANUFACTURER.

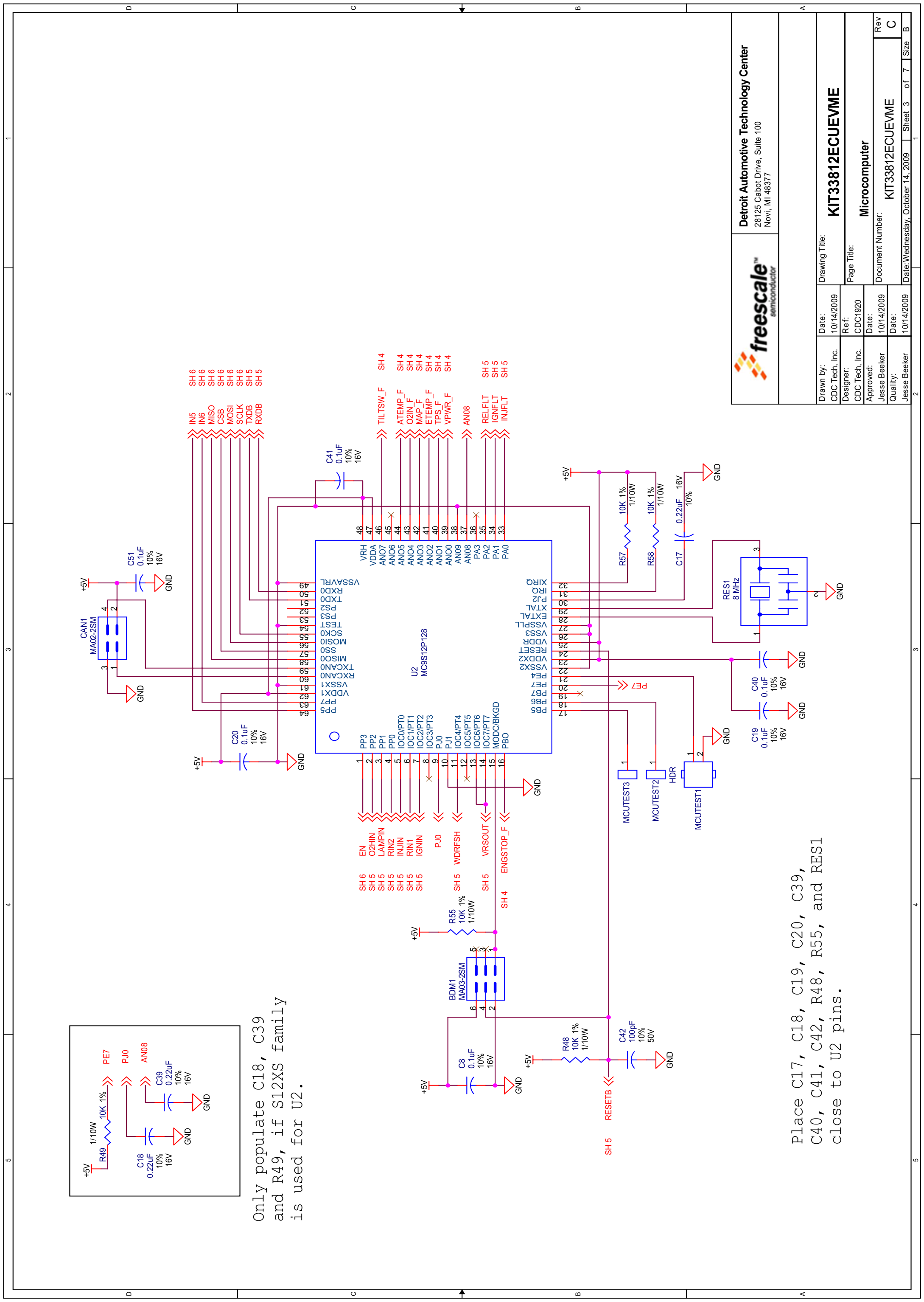
4. SPECIAL SIGNAL USAGE:
_B DENOTES - ACTIVE-LOW SIGNAL
<> OR [] DENOTES - VECTORED SIGNALS

5. INTERPRET DRAWING IAW ANSI SPECIFICATIONS, CURRENT REVISION, WITH THE EXCEPTION OF LOGIC BLOCK SYMBOLOGY.

freescale semiconductor

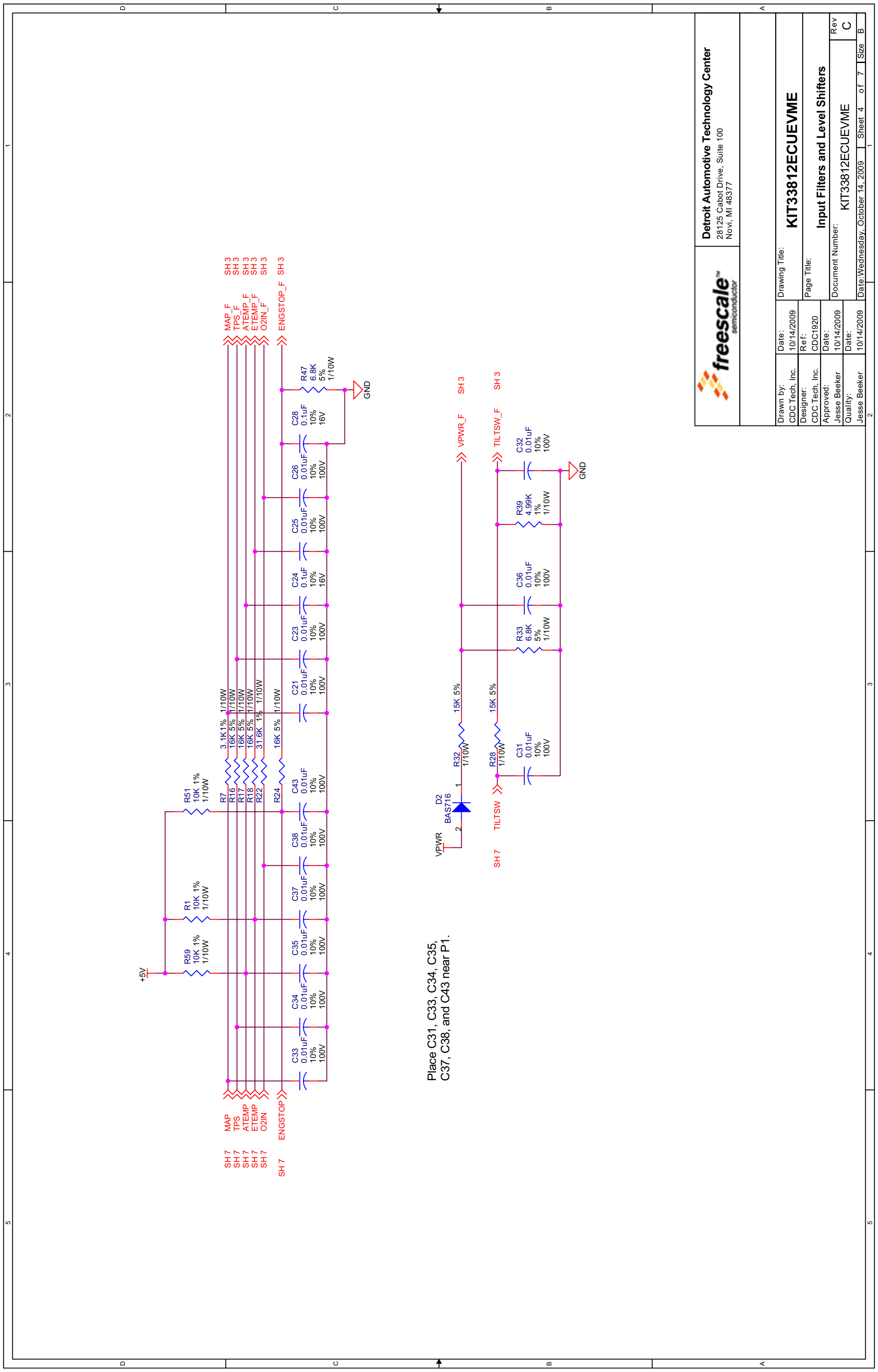
Detroit Automotive Technology Center
28125 Cabot Drive, Suite 100
Novi, MI 48377

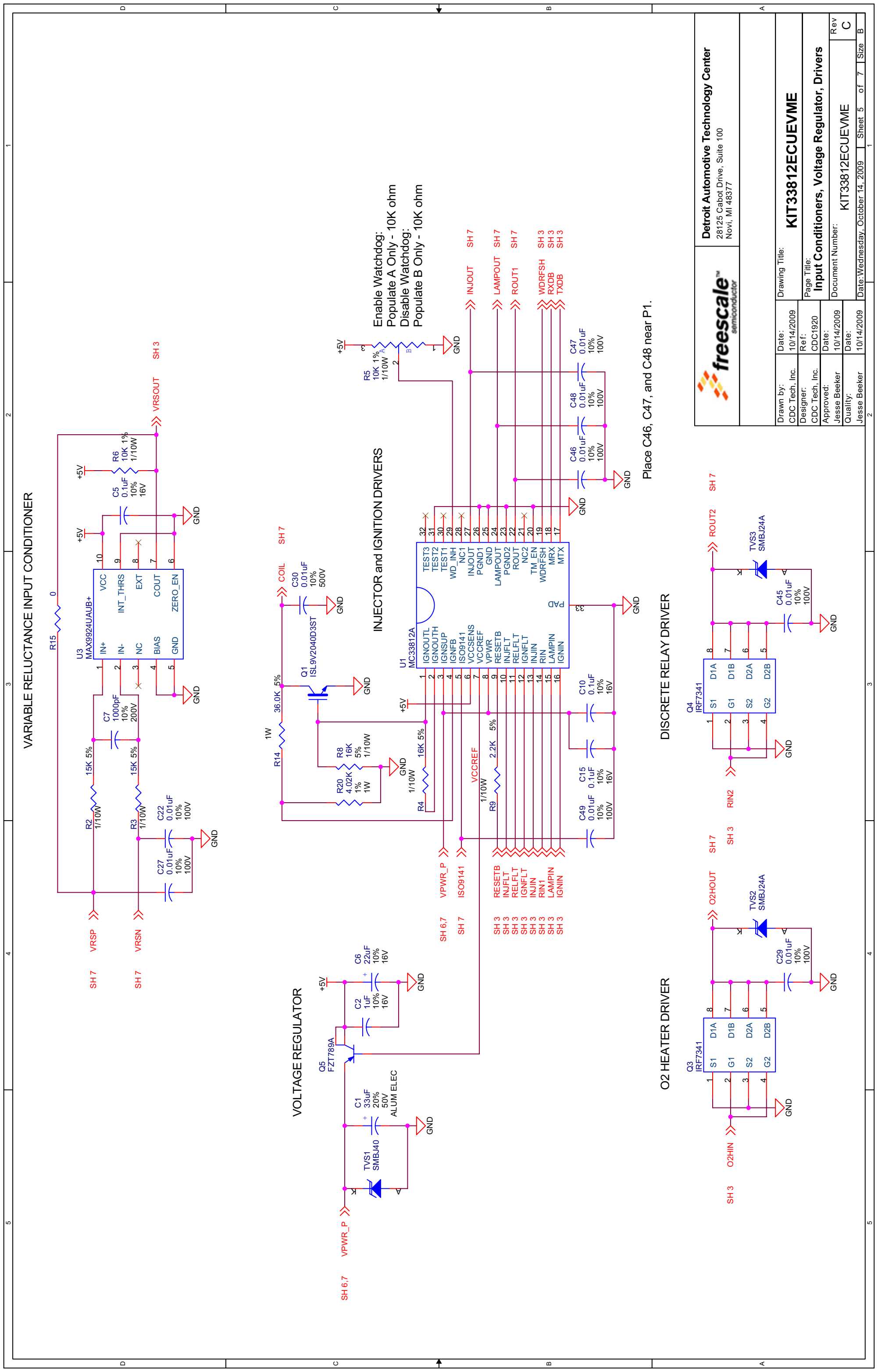
Drawn by: CDC Technologies, Inc.	Date: 10/14/2009	TITLE: KIT33812ECUEVME			
Designer: CDC Technologies, Inc.	Ref: CDC1920				
Approved: Jesse Beeker	Date: 10/14/2009	SIZE B	GEDTTL: GEDABV:	DWG. NO. KIT33812ECUEVME	REV: C
Quality: Jesse Beeker	Date: 10/14/2009	LAST MODIFIED= Wednesday, October 14, 2009			SHEET 1 of 7

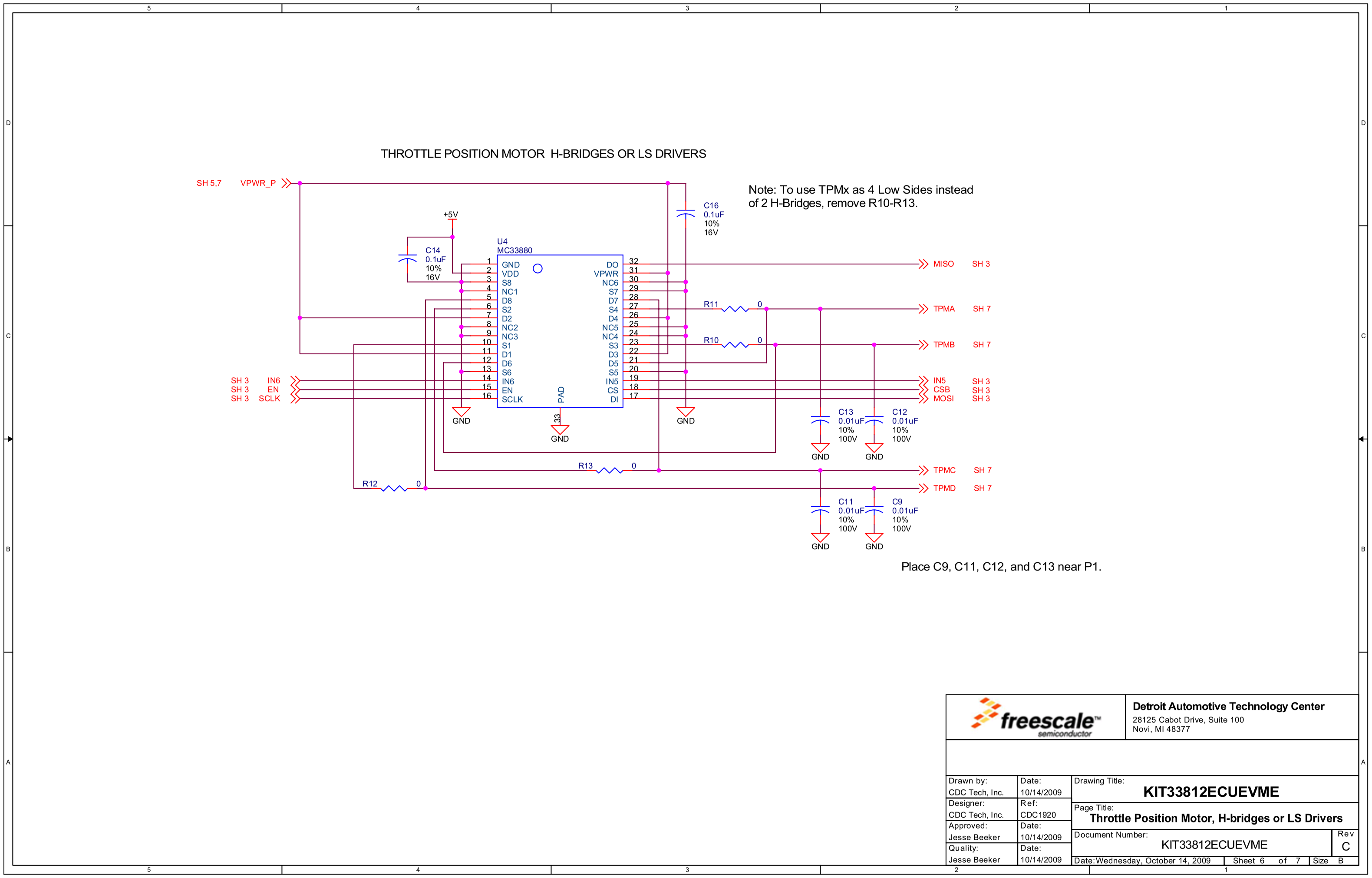


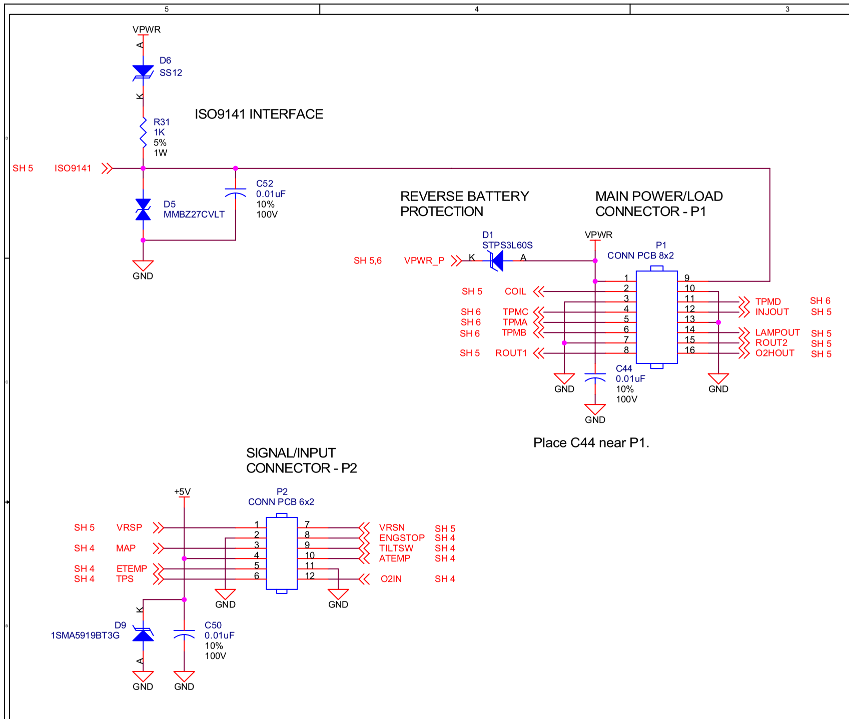
Only populate C18, C39 and R49, if S12XS family is used for U2.

Place C17, C18, C19, C20, C39, C40, C41, C42, R48, R55, and RES1 close to U2 pins.


Detroit Automotive Technology Center
28125 Cabot Drive, Suite 100
Novi, MI 48377
freescale™ semiconductor
Drawn by: CDC Tech, Inc.
Date: 10/14/2009
Designer: CDC Tech, Inc.
Ref: CDC1920
Approved: Jesse Beeker
Date: 10/14/2009
Quality: Jesse Beeker
Date: 10/14/2009
Drawing Title: KIT33812ECUEVME
Page Title: Input Filters and Level Shifters
Document Number: KIT33812ECUEVME
Rev C
Date: Wednesday, October 14, 2009
Sheet 4 of 7
Size B
SH 7 MAP
SH 7 TPS
SH 7 ATEMP
SH 7 ETEMP
SH 7 O2IN
SH 7 ENGSTOP
MAP_F SH 3
TPS_F SH 3
ATEMP_F SH 3
ETEMP_F SH 3
O2IN_F SH 3
ENGSTOP_F SH 3
+5V
R59 10K 1% 1/10W
R1 10K 1% 1/10W
R51 10K 1% 1/10W
R7 3.1K 1% 1/10W
R16 16K 5% 1/10W
R17 16K 5% 1/10W
R18 16K 5% 1/10W
R22 31.6K 1% 1/10W
R24 16K 5% 1/10W
C33 0.01uF 10% 100V
C34 0.01uF 10% 100V
C35 0.01uF 10% 100V
C37 0.01uF 10% 100V
C38 0.01uF 10% 100V
C43 0.01uF 10% 100V
C21 0.01uF 10% 100V
C23 0.01uF 10% 100V
C24 0.1uF 10% 16V
C25 0.01uF 10% 100V
C26 0.01uF 10% 100V
C28 0.1uF 10% 16V
R47 6.8K 5% 1/10W
GND
VPWR
D2 BAS716
2
1
R32 1/10W 15K 5%
VPWR_F SH 3
SH 7 TILTSW
R28 1/10W 15K 5%
TILTSW_F SH 3
C31 0.01uF 10% 100V
R33 6.8K 5% 1/10W
C36 0.01uF 10% 100V
R39 4.99K 1% 1/10W
C32 0.01uF 10% 100V
GND
Place C31, C33, C34, C35, C37, C38, and C43 near P1.
A
B
C
D
1
2
3
4
5


VARIABLE RELUCTANCE INPUT CONDITIONER
U3
MAX9924UAUB+
IN+
IN-
NC
BIAS
GND
VCC
INT_THRS
EXT
COUT
ZERO_EN
R15 0
R2 15K 5% 1/10W
R3 15K 5% 1/10W
C7 1000pF 10% 200V
C22 0.01uF 10% 100V
C27 0.01uF 10% 100V
C5 0.1uF 10% 16V
R6 10K 1% 1/10W
VRSOUT
VRSP
VRSN
SH 3
SH 7
+5V
GND
INJECTOR and IGNITION DRIVERS
U1
MC33812A
GNOUTL
GNOUTH
IGNSUP
IGNFB
ISO9141
VCCSENS
VCCREF
VPWR
RESETB
INJFLT
RELFLT
IGNFLT
INJIN
RIN
LAMPIN
IGNIN
TEST3
TEST2
TEST1
WD_INH
NC1
INJOUT
PGND1
GND
LAMPOUT
PGND2
ROUT
NC2
TM_EN
WDRFSH
MRX
MTX
PAD
Q1
ISL9V2040D3ST
COIL
C30 0.01uF 10% 500V
R14 36.0K 5% 1W
R8 16K 5% 1/10W
R20 4.02K 1% 1W
R4 16K 5% 1/10W
R9 2.2K 5% 1/10W
VCCREF
C10 0.1uF 10% 16V
C15 0.1uF 10% 16V
C49 0.01uF 10% 100V
C46 0.01uF 10% 100V
C48 0.01uF 10% 100V
C47 0.01uF 10% 100V
R5 10K 1% 1/10W
Enable Watchdog:
Populate A Only - 10K ohm
Disable Watchdog:
Populate B Only - 10K ohm
INJOUT
LAMPOUT
ROUT1
WDRFSH
RXDB
TXDB
VPWR_P
ISO9141
RESETB
INJFLT
RELFLT
IGNFLT
INJIN
RIN1
LAMPIN
IGNIN
SH 6,7
Place C46, C47, and C48 near P1.
VOLTAGE REGULATOR
Q5
FZT789A
C6 22uF 10% 16V
C2 1uF 10% 16V
C1 33uF 20% 50V ALUM ELEC
TVS1 SMBJ40
VPWR_P
DISCRETE RELAY DRIVER
Q4
IRF7341
S1
G1
S2
G2
D1A
D1B
D2A
D2B
TVS3 SMBJ24A
C45 0.01uF 10% 100V
ROUT2
RIN2
O2 HEATER DRIVER
Q3
IRF7341
TVS2 SMBJ24A
C29 0.01uF 10% 100V
O2HOUT
O2HIN
Detroit Automotive Technology Center
28125 Cabot Drive, Suite 100
Novi, MI 48377
freescale semiconductor
Drawn by: CDC Tech, Inc.
Designer: CDC Tech, Inc.
Approved: Jesse Beeker
Quality: Jesse Beeker
Date: 10/14/2009
Ref: CDC1920
Date: 10/14/2009
Date: 10/14/2009
Drawing Title:
KIT33812ECUEVME
Page Title:
Input Conditioners, Voltage Regulator, Drivers
Document Number:
KIT33812ECUEVME
Rev C
Date: Wednesday, October 14, 2009
Sheet 5 of 7
Size B


THROTTLE POSITION MOTOR H-BRIDGES OR LS DRIVERS
Note: To use TPMx as 4 Low Sides instead of 2 H-Bridges, remove R10-R13.
Place C9, C11, C12, and C13 near P1.
SH 5,7
VPWR_P
+5V
C14
0.1uF
10%
16V
C16
0.1uF
10%
16V
U4
MC33880
1 GND
2 VDD
3 S8
4 NC1
5 D8
6 S2
7 D2
8 NC2
9 NC3
10 S1
11 D1
12 D6
13 S6
14 IN6
15 EN
16 SCLK
32 DO
31 VPWR
30 NC6
29 S7
28 D7
27 S4
26 D4
25 NC5
24 NC4
23 S3
22 D3
21 D5
20 S5
19 IN5
18 CS
17 DI
PAD
33
GND
R11 0
R10 0
R13 0
R12 0
MISO SH 3
TPMA SH 7
TPMB SH 7
IN5 SH 3
CSB SH 3
MOSI SH 3
TPMC SH 7
TPMD SH 7
SH 3 IN6
SH 3 EN
SH 3 SCLK
C13
0.01uF
10%
100V
C12
0.01uF
10%
100V
C11
0.01uF
10%
100V
C9
0.01uF
10%
100V
freescale semiconductor
Detroit Automotive Technology Center
28125 Cabot Drive, Suite 100
Novi, MI 48377
Drawn by: CDC Tech, Inc.
Date: 10/14/2009
Designer: CDC Tech, Inc.
Ref: CDC1920
Approved: Jesse Beeker
Date: 10/14/2009
Quality: Jesse Beeker
Date: 10/14/2009
Drawing Title: KIT33812ECUEVME
Page Title: Throttle Position Motor, H-bridges or LS Drivers
Document Number: KIT33812ECUEVME
Rev C
Date: Wednesday, October 14, 2009
Sheet 6 of 7
Size B
1
2
3
4
5
A
B
C
D



P1	1	VPWR	Power Input	13.8V	System power from 12V battery
	2	COIL	Output with feedback	0-Vbat	Spark control of digital ignition system.
	3	GND	Power Output	0V	Module level ground reference, return path of Vbat
	4	TPMC	Output with feedback	0-Vbat	H-bridge control for 4-phase stepper motor for idle air speed control.
	5	TPMA	Output with feedback	0-Vbat	H-bridge control for 4-phase stepper motor for idle air speed control.
	6	TPMB	Output with feedback	0-Vbat	H-bridge control for 4-phase stepper motor for idle air speed control.
	7	GND	Power Output	0V	Module level ground reference, return path of Vbat
	8	ROUT1	Output with feedback	0-Vbat	Relay driver output
	9	ISO9141	Input and Output	0-Vbat	Bi-direction communication pin for diagnostics.
	10	GND	Power Output	0V	Module level ground reference, return path of Vbat
	11	TPMD	Output with feedback	0-Vbat	H-bridge control for 4-phase stepper motor for idle air speed control.
	12	INJOUT	Output with feedback	0-Vbat	Fuel injector control.
	13	GND	Power Output	0V	Module level ground reference, return path of Vbat
	14	LAMPOUT	Output	0-Vbat	Incandescent light bulb control
	15	ROUT2	Output	0-Vbat	Relay driver output
	16	O2HOUT	Output	0-Vbat	Oxygen sensor heater element

P2	1	VRSP	Analog Input	0-100V	Positive connection to variable reluctance sensor used for crankshaft position. Alternatively used as input for hall type sensor.
	2	GND	Power Output	0V	Module level ground reference, return path of Vbat
	3	MAP	Analog Input	0-5V	Sensor input indicating the pressure on the intake manifold of the engine
	4	VREF	Analog Output	5V	5V system reference
	5	ETEMP	Analog Input	0-5V	Sensor input indicating the current temperature of the engine block or coolant
	6	TPS	Analog Input	0-5V	Throttle position sensor indicating the current state of the butterfly in of the throttle body
	7	VRSN	Analog Input	0-100V	Negative connection to variable reluctance sensor used for crankshaft position
	8	ENGSTOP	Digital Input	0-Vbat	Signal indicating the state of the engine shutoff switch
	9	TILTSW	Analog or Digital Input	0-5V	Signal indicating that the engine is in a safe orientation to run.
	10	ATEMP	Analog Input	0-5V	Sensor input indicating outside air temperature
	11	GND	Power Output	0V	Module level ground reference, return path of Vbat
	12	O2IN	Analog Input	0-5V	Oxygen sensor input indicating mixture conditions from exhaust

PCB FABRICATION NOTES:

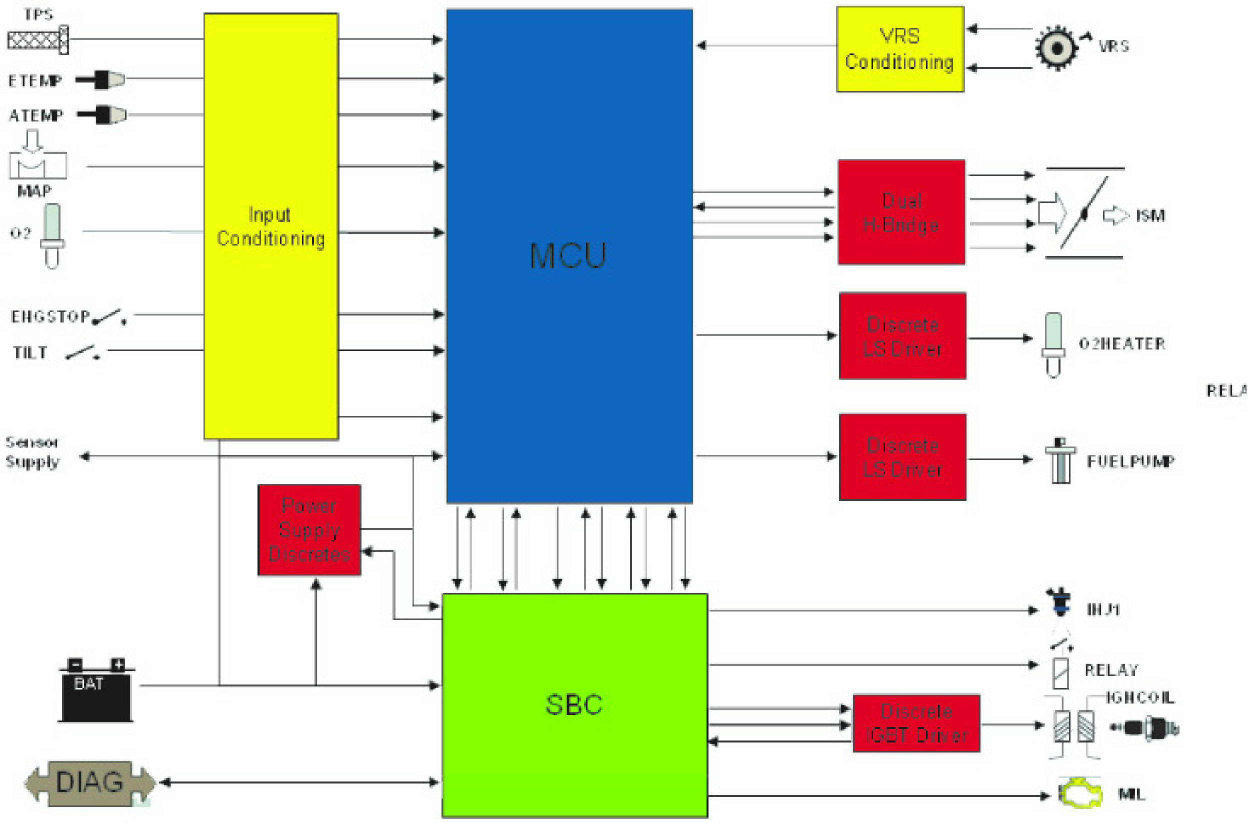
1. FINISHED BOARD SIZE: 2.0" X 3.5"
2. FINISHED BOARD THICKNESS: 0.062"
3. SURFACE FINISH: IMMERSION GOLD
4. BOARD SUBSTRATE MATERIAL: FR406
5. SOLDER MASK: LPI COLOR BLUE BOTH SIDES
6. SILKSCREEN: WHITE EPOXY INK, BOTH SIDES
7. FABRICATION QUALITY, IPC-A-600 CLASS 2
8. DRILL HOLE LOCATIONS: +/-0.003" FROM TRUE
9. FINISHED HOLE SIZE: +/-0.003"

freescale semiconductor

Detroit Automotive Technology Center
28125 Cabot Drive, Suite 100
Novi, MI 48377

Drawn by: CDC Tech, Inc.	Date: 10/14/2009	Drawing Title: KIT33812ECUEVME	
Designer: CDC Tech, Inc.	Ref: CDC1920	Page Title: ISO9141 Interface, Connector	
Approved: Jesse Beeker	Date: 10/14/2009	Document Number: KIT33812ECUEVME	Rev C
Quality: Jesse Beeker	Date: 10/14/2009	Date:Wednesday, October 14, 2009 Sheet 7 of 7	Size B

8 System Block Diagram



9 Bill of Materials

Ref Number	Qty	Part Type	Value	Vendor/Source	Part Number	Package Type
C1	1	Capacitor, Elect. Alum.	33uF 50V 20%	Panasonic - ECG Digi-Key	EEE-HA1H330XP PCE4218TR-ND	Radial - Surface Mount
C2	1	Capacitor, Ceramic	1.0uF 16V 10% X7R	Taiyo Yuden Digi-Key	EMK107B7105KA-T 587-1241-1-ND	0603 (1608 metric)
C5,C8,C10, C14-C16,C19,C20, C24,C28,C40-C41,**C51**	13	Capacitor, Ceramic	0.1uF 16V 10% X7R	Panasonic - ECG Digi-Key	ECJ-1VB1C104K PCC1762TR-ND	0603 (1608 metric)
C6	1	Capacitor, Tantalum	22uF 16V 10%	Kemet Digi-Key	B45196H3226K209 495-2242-1-ND	SMT3528-21 (EIA-B)
C7	1	Capacitor, Ceramic	1000pF 200V 10% X7R	AVX Corporation Digi-Key	06032C102KAT2A 478-1194-1-ND	0603 (1608 metric)
C9,C11-13,C21,**C22**, C23,C25,C26,**C27**,C29, C31-C38,C43-C50,C52	28	Capacitor, Ceramic	0.01uF 100V 10% X7R	Murata Digi-Key	GCM188R72A103KA37D 490-4781-1-ND	0603 (1608 metric)
C30	**1**	**Capacitor, Ceramic**	**0.01uF 500V 10%**	Digi-Key		**1206 (3216 metric)**
C17,C18**,C39**	3	Capacitor, Ceramic	0.22uF 16V 10% X7R	Taiyo Yuden	EMK107B7224KA-T	0603 (1608 metric)
				Digi-Key	587-1249-2-ND	
C42	1	Capacitor, Ceramic	100pF 50V 10% NP0	AVX Corporation Digi-Key	06035A101KAT2A 478-3717-2-ND	0603 (1608 metric)
R1,R5***,R6,R48, R49**,R51,R55,R57-59	10	Resistor	10.0K-Ohm 1/10W 1%	Yageo Digi-Key	RC0603FR-0710KL 311-10.0KHRCT-ND	0603 (1608 metric)
R2,R3,R28,R32	4	Resistor	15.0K-Ohm 1/10W 5%	Yageo Digi-Key	RC0603JR-0715KL 311-15KGRCT-ND	0603 (1608 metric)
R4,R8,R16-18,R24	6	Resistor	16.0K-Ohm 1/10W 5%	Yageo Digi-Key	RC0603JR-0716KL 311-16KGRCT-ND	0603 (1608 metric)
R7	1	Resistor	3.09K-Ohm 1/10W 1%	Yageo Digi-Key	RC0603FR-073K09L 311-3.09KHRCT-ND	0603 (1608 metric)
R9	1	Resistor	2.2K-Ohm 1/10W 5%	Yageo Digi-Key	RC0603JR-072K2L 311-2.2KGRCT-ND	0603 (1608 metric)
R10-13,**R15**	5	Resistor	0-Ohm 1/8W	Stackpole Electronics Inc Digi-Key	RMCF 1/10 0 R RMCF1/100RTR-ND	0805 (2012 metric)
R14	1	Resistor	36.0K-Ohm 1W 5%	Vishay/Dale Digi-Key	CRCW251236K0JNEG 541-36KXTR-ND	2512 (6432 metric)
R20	1	Resistor, Thick Film	4.02K-Ohm 1W 1%	Vishay/Dale Digi-Key	CRCW25124K02FKEG 541-4.02KAFTR-ND	2512 (6432 metric)
R22	1	Resistor	31.6K-Ohm 1/10W 1%	Yageo Digi-Key	RC0603FR-0731K6L 311-31.6KHRCT-ND	0603 (1608 metric)

Ref Number	Qty	Part Type	Value	Vendor/Source	Part Number	Package Type
R31	1	Resistor, Thick Film	1.0K-Ohm 1/8W 5%	Rohm Digi-Key	KTR10EZPJ102 RHM1.0KBCT-ND	0805 (2012 metric)
R33,R47	2	Resistor	6.8K-Ohm 1/10W 5%	Yageo Digi-Key	RC0603JR-076K8L 311-6.8KGRCT-ND	0603 (1608 metric)
R39	1	Resistor	4.99K-Ohm 1/10W 1%	Yageo Digi-Key	RC0603FR-074K99L 311-4.99KHRCT-ND	0603 (1608 metric)
CAN1	**1**	**Header**	**2x2 0.100" Pitch**	**Sullins** Digi-Key	**GEC02DABN-M30** **S2331E-02-ND**	**SMT Special**
BDM1	1	Header	2x3 0.100" Pitch	FCI Digi-Key	95278-101A06LF 609-3487-2-ND	SMT Special
P1	1	Connector 16 pos	2x8	Tyco Electronics Digi-Key	1-1586039-6 A30675-ND	Through-Hole
P2	1	Connector 12 pos	2x6	Tyco Electronics Digi-Key	1-1586039-2 A30673-ND	Through-Hole
J1	2	Plug 16 pos	2x8	Tyco Electronics Digi-Key	1-794954-6 A30597-ND	Through-Hole
J2	2	Plug 12 pos	2x6	Tyco Electronics Digi-Key	1-794954-2 A30595-ND	Through-Hole
Terminal	28	Plug Terminal	18-22AWG crimp	Tyco Electronics Digi-Key	1586315-4 A30645-ND	Wire Crimp
D1	1	Diode, Schottky	60V 3A	STMicroelectronics Digi-Key	STPS3L60S 497-2454-2-ND	DO-214AB, SMC
D2, D6	2	Diode, Schottky	20V 1A	Vishay/General Semiconductor Digi-Key	SS12-E3/61T SS12-E3/61TGITR-ND	DO-214AC, SMA
D5	1	Diode, Zener	27V 40W	On Semi Digi-Key	MMBZ27VCLT1G MMBZ27VCLT1GOSCT-ND	SOT-23
D9	**1**	**Diode, Zener**	**5.6V 1.5W**	**On Semi** Digi-Key	**1SMA5919BT3G** **1SMA5919BT3GOSCT-ND**	**DO-214AC, SMA**
Q1	1	Transistor, IGBT	400V 10A	Fairchild Semiconductor Digi-Key	ISL9V2040D3ST ISL9V2040D3STTR-ND	TO252AA
Q3,Q4	2	Transistor, 2N	55V 4.7A	International Rectifier Digi-Key	IRF7341TRPBF IRF7341PBFTR-ND	SOIC-8
Q5	1	Transistor, PNP	25V 3A	Zetex Inc. Digi-Key	FZT789ATA FZT789ATR-ND	SOT-223
TVS1	1	Diode, TVS	22V 1500W Unidir.	Littelfuse Inc Digi-Key	SMCJ22A SMCJ22ALFCT-ND	DO-214AB, SMC
TVS2,TVS3	2	Diode, TVS	40V 600W Unidir.	Diodes Inc. Digi-Key	SMBJ40A-13-F SMBJ40A-FDITR-ND	DO-214AA, SMB
U1	1	IC	1-Cyl. Phys. Layer	Freescale Semiconductor	MC33812	SOIC-32
U2	1	IC	16 Bit MCU	Freescale Semiconductor	MC9S12XS64MAE	LQFP-64
U2 Alternative	1	IC	16 Bit MCU	Freescale Semiconductor	MC9S12P128	LQFP-64
U3	1	IC	Variable Reluctance Sensor Interface	Maxim Digi-Key	MAX9924UAUB+T MAX9924UAUB+-ND	10uMax

Ref Number	Qty	Part Type	Value	Vendor/Source	Part Number	Package Type
U4	1	IC	Configurable Driver	Freescale Semiconductor	MCZ33880EW	SOIC-32
U4 Alternative	1	IC	Configurable Driver	Freescale Semiconductor	MCZ33879EK	SOIC-32
				Freescale Semiconductor	MCZ33879EK	
RES1	1	Resonator	8.0 Mhz Resonator	Murata Digi-Key	CSTCE8M00G55-R0 490-1195-2-ND	Custom SMT
	1	Circuit Board		DDi	CDC1796	PCB

Bold items are not populated.

ASSEMBLY NOTES:

**C18,C39,R49 only populated when U2=MC9S12XS64

***R5 populated on pads R5:2-3

10 Appendix A: Hardware Reference Manual

The KIT33812ECUEVME is a working reference design for small engine control. It can be used right out of the box to test and development applications that run small internal combustion engines, such as those found on generators, lawn mowers, and motorbikes. The system is designed to be cost efficient for a one-cylinder type of engine. Use of this design is an excellent starting point towards making a custom small engine controller. Modification of the design was taken into consideration and basic information is provided as a consideration for an application specific, production engine controller. This document will review the key features of KIT33812ECUEVME and present design level information on the key circuits in the design.

10.1 Background

As the government regulations around the world begin imposing regulations on small engines, electronic fuel injection is the building block for the future of motorbikes, ATVs, lawn mowers, and all other low displacement engines. Unlike large displacement engines of the automotive market, the change from carburetion to electronic control has not been fully adopted. The main reasons for the resistance to change are: cost and packaging. Small engines have been mechanically refined for over 20 years. This does not leave much room for adding electronic components. This is especially true when the additional electronic hardware adds significant cost to the product.

10.2 System Design

To address these concerns of the small engine market, the KIT33812ECUEVME is designed to be very efficient in cost and size. Additionally the ECU is able to aid in the development of a next generation small engine capable of meeting new emissions regulations. The key to creating such a system is the use of integrated components specific to one-cylinder control. As a basis for the design, the Freescale MC33812 Small Engine Control IC was used. This device provides extensive integration of key components that are essential to small engine control.

The KIT33812ECUEVME also takes advantage of the cost efficient MC9S12P128 microcontroller from Freescale. This 16-bit processor rivals the cost of 8-bit devices while providing the right amount of performance for a one-cylinder application. It also allows upward compatibility into the S12X MCU product family for higher demand applications. Use of a 16-bit processor instead of a 32-bit processor keeps the system costs down. While this limits the maximum performance of the system, less overall performance is required due to the reduced complexity of a small engine compared to a 4-cylinder or larger Automotive application.

A very specific set of signals was selected for the KIT33812ECUEVME. The complete list of signals for the system is provided in **Figure 22**. These signals reflect a system capable of closed-loop control of an engine with the potential to meet at Euro III and above emissions levels. A complete list of the signals with functionality and DC specifications is provided in Table “DC Specifications of the Signals for KIT33812ECUEVME”

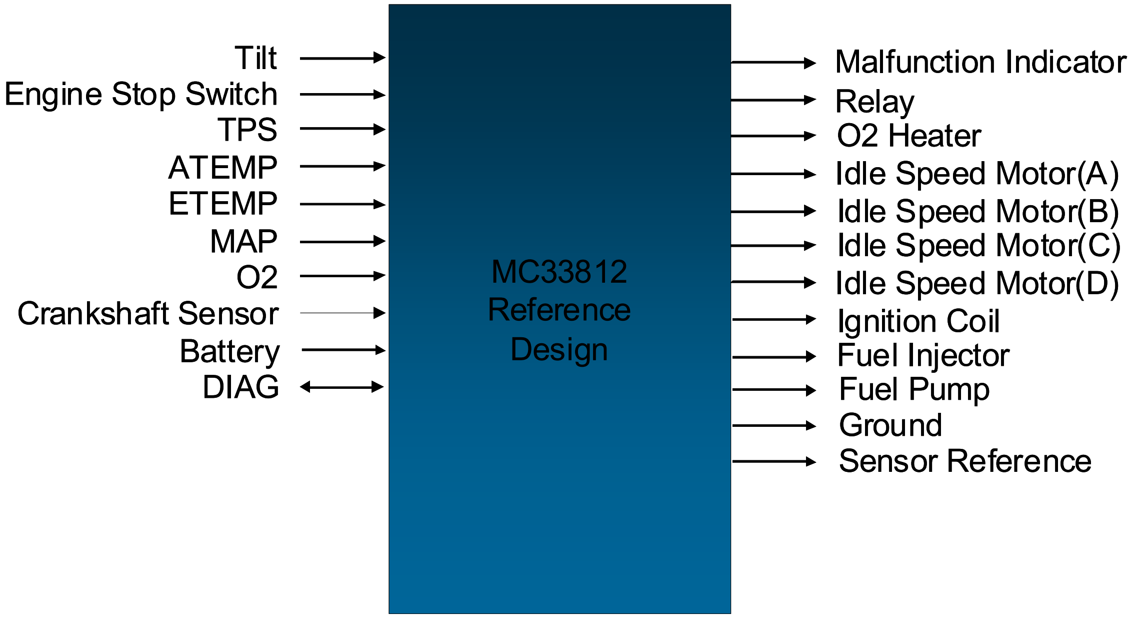


Figure 22. Signals Diagram for the KIT33812ECUEVME Small Engine Reference Design

.

DC Specifications of the Signals for KIT33812ECUEVME

KIT33812ECUEVME Reference Design					
Connector	Pin	Signal Name	Signal Type	Voltage Range	Recommended Functionality
P1	1	VPWR	Power Input	14 V	System power from 12V battery
	2	COIL	Output with feedback	0-14 V	Spark control of digital ignition system.
	3	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	4	TPMC	Output with feedback	0-14 V	H-bridge control for 4-phase stepper motor for idle air speed control
	5	TPMA	Output with feedback	0-14 V	H-bridge control for 4-phase stepper motor for idle air speed control
	6	TPMB	Output with feedback	0-14 V	H-bridge control for 4-phase stepper motor for idle speed air speed control
	7	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	8	ROUT1	Output with feedback	0-14 V	Relay driver output
	9	ISO9141	Input and Output	0-14 V	Bi-direction communication pin for diagnostics
	10	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	11	TPMD	Output with feedback	0-14 V	H-bridge control for 4-phase stepper motor for idle air speed control
	12	INJOUT	Output with feedback	0-14 V	Fuel injector control
	13	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	14	LAMPOUT	Output	0-14 V	Incandescent light bulb control
	15	ROUT2	Output	0-14 V	Relay driver output
	16	O2HOUT	Output	0-14 V	Oxygen sensor heater element

DC Specifications of the Signals for KIT33812ECUEVME

KIT33812ECUEVME Reference Design

Connector	Pin	Signal Name	Signal Type	Voltage Range	Recommended Functionality
P2	1	VRSP	Analog Input	0-100 V	Positive connection to variable reluctance sensor used for crankshaft position. Alternatively used as input for hall type sensor
	2	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	3	MAP	Analog Input	0-5 V	Sensor input indicating the pressure on the intake manifold of the engine
	4	VREF	Analog Output	5 V	5V system reference
	5	ETEMP	Analog Input	0-5 V	Sensor input indicating the current temperature of the engine block or coolant
	6	TPS	Analog Input	0-5 V	Throttle position sensor indicating the current state of the butterfly in of the throttle body
	7	VRSN	Analog Input	0-100 V	Negative connection to variable reluctance sensor used for crankshaft position
	8	ENGSTOP	Digital Input	0-14 V	Signal indicating the state of the engine shutoff switch
	9	TILTSW	Analog or Digital Input	0-5 V	Signal indicating that the engine is in a safe orientation to run
	10	ATEMP	Analog Input	0-5 V	Sensor input indicating outside air temperature
	11	GND	Power Output	0 V	Module level ground reference, return path of Vbat
	12	O2IN	Analog Input	0-5 V	Oxygen sensor input indicating mixture conditions from exhaust

10.3 Key Circuit Identification

A total of 8 key circuits have been identified in the KIT33812ECUEVME. Each of these circuits has been specifically designed to as a starting point for designing a custom electronic control unit. It is recommended to change or validate these circuits when implementing a production ECU or a custom design. This will produce a higher level of safety and a more cost efficient design as the KIT33812ECUEVME was not designed to go into a specific application beyond a generic one-cylinder engine implementation. Figure 23 identifies the key circuits in the system.

10.4 KIT33812ECUEVME System Block Diagram

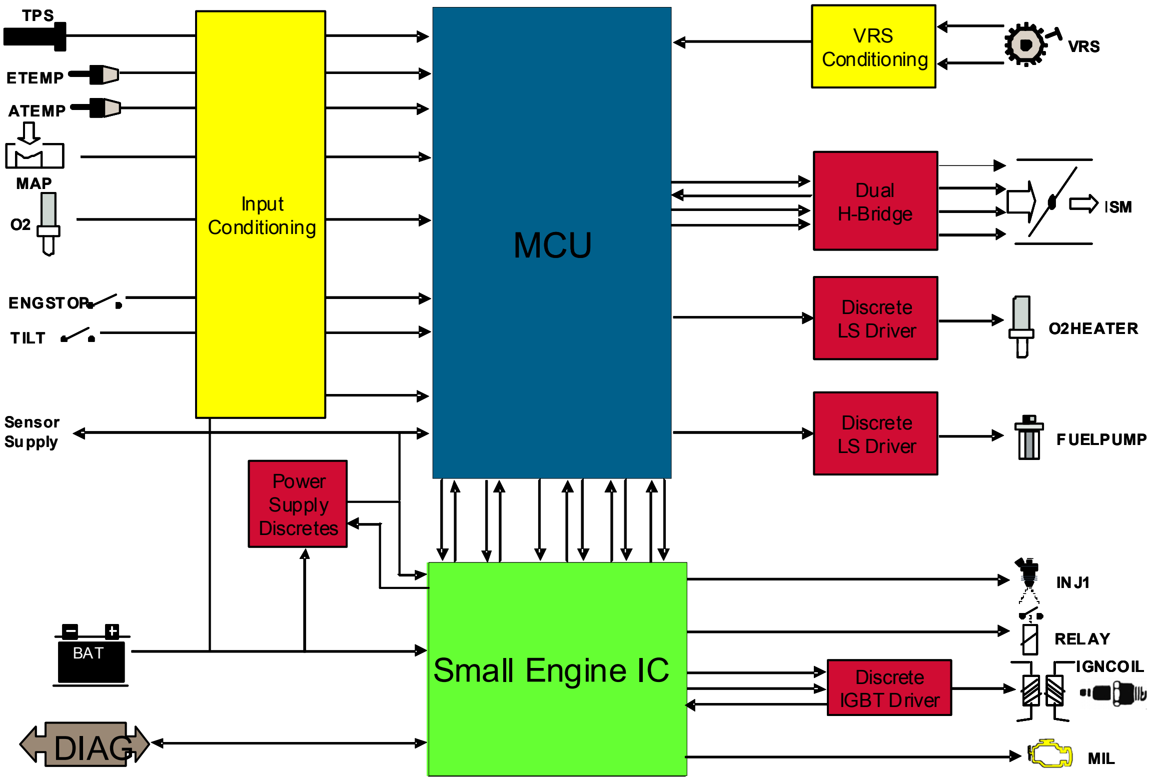


Figure 23. System Block Diagram of KIT33812ECUEVME and Example System Loads

10.4.1 MCU

10.4.1.1 Functional Description

All application control is provided by the MCU. Through indirect connections to the engine system, user controls are interpreted and fuel and spark events are generated by the MCU. User software implements control algorithms based on hardware level features of the MCU.

10.4.1.2 Design Criteria

Using the maximum RPM of the application, an estimation of the required performance of the MCU can be created. For a small engine, the 10kRPM operating point is a strong possibility with certain applications going beyond. To get a feel for how much processing is required for an application, benchmarks of critical tasks are typically used. While no specific benchmark for small engine control is available, the ability to process 2000 instructions between two consecutive teeth on a 24 minus 1 crankshaft flywheel was used. This number is approximately 2 times as many cycles as necessary for the critical processing and initiation of a spark event in the example application used with this reference design. Use of 2000 cycles represents exceptional performance for critical processing tasks.

For the case of the 10 kRPM operating point and the 24 minus 1 toothed-wheel, these 2000 instructions must execute in 25 μs or the risk of severe system latencies can cause the application to lose synchronization of the rotating engine, resulting in loss of control. If an instruction is processed by the core of the MCU on every bus clock cycle (RISC architecture), this equates to 8MHz performance. The selection of the MC9S12P128 MCU exceeds this demand by providing a maximum processing speed of 32 MHz, based on its CISC architecture and an average of 3 cycles per instruction.

In addition to processing power, key hardware peripherals are required for engine control. These include: input capture, output compare, PWM, A/D converter, communication port, and general purpose I/O. Input capture is specifically used to process pulses from the toothed-wheel. Output compare is used to generate fuel and spark events. Specifically, input capture and output compare functions must use the same timer and a time base that allows a dynamic range of operation for an engine control application. This means that at high RPM, the timer counter must be able to distinguish events and at low RPM there must be enough counts to not overflow. PWM generation is essential for power management of loads. This is a key concept for a small engine as electrical efficiency provides dramatic improvements in operating performance. Sensor measurements are obtained by the A/D converter. Communication outside the module is required for diagnostics and possible control or data output to other modules in the system.

The MC9S12P128 meets these performance demands with a multi-channel 16-bit timer, PWM generator, multi-channel 12 bit A/D converter, CAN and SCI ports, and large number of I/O. These features are complemented by internal FLASH memory and the single wire BDM debugging/programming port that aid in the development and deployment of software. Small package options from 48 to 80 pins, memory sizes from 32 k to 128 k, and strong compatibility to the S12X product family add flexibility for implementing a custom design.

10.4.1.3 Implementation Recommendations

Designing a system with a microcontroller takes experience. MCU manufacturers have specific guidelines and recommendations that should be used as a starting point towards a successful design. This primarily concern power supply bypassing and the MCU hardware configuration pins. Guidelines provided in the datasheet for the MC9S12P128 in the 64 pin package have been demonstrated in this design. Additionally, considerations for a 2 layer printed circuit board have been implemented that reflect the high frequency operation large load switching in the system. This is reflected in the routed power traces and comprehensive ground return paths for each signal.

The cost sensitivity of a small engine controller is reflected by the design decision to use a resonator and not a crystal for the oscillator. Resonators offer significant cost reduction when compared to a crystal, but at the sacrifice of precision. The resonator selected is relatively high tolerance and satisfies typical OEM criteria for the generation of CAN communication bit timing for medium to high speed data rates. With the MC9S12P128 MCU, further cost reduction can be explored by the making use of the internal reference clock. This would eliminate the use of an external oscillator but would significantly reduce the timing performance of the system. Additional cost reductions to the MCU circuit would include the use of the 48 pin QFN package.

Consideration for the S12XS product family was addressed in the design. This results in a low number of additional components that can be placed onto the PCB when using the S12XS products in the 64 pin package. Using this MCU will allow increased performance and increased memory size if the application demands it. Significantly higher performance and memory size can be achieved through implementing an S12XE family processor. Minimal hardware changes would be required to move the design to the S12XE, which provides up to 50 MHz operation and adds the XGATE co-processor to off load the main processor.

10.4.2 Small Engine Control IC

10.4.2.1 Functional Description

The Small Engine Control IC (SECIC) has the ability to control a core set of loads specific to a small engine. For the KIT33812ECUEVME, a single cylinder application is targeted with the following loads identified as critical: fuel injector driver, ignition coil pre-driver, system voltage regulator, relay driver, lamp driver, system watchdog, voltage reset generator, and diagnostic communication physical layer. Control signals from the MCU are interpreted by the SECIC to drive the critical loads for the small engine system.

Fuel delivery through a fuel injector is performed by direct connection to the injector driver. Spark control for a transistor controlled, inductive (TCI) ignition is accomplished through a pre-driver output paired with an external IGBT. System voltage is created by an integrated 5V pre-regulator that uses an external pass-transistor to off-load power dissipation. Relay and lamp driver output directly control loads. An optional system watchdog is integrated with a flexible programming time and optional hardware disable. Power on control and system voltage monitoring is integrated and capable of providing a system reset for loss of power and brownout conditions. Communication to external electronic modules, such as a diagnostic tool, is provided by a single wire ISO-9141 transceiver.

10.4.2.2 5 Volt Regulator Pre-driver

10.4.2.2.1 Design Criteria

This functional block steps down the system battery voltage to the 5 volt level required by the MCU, sensors, and other circuit elements. The VCCREF pin provides the base current drive output for an external PNP pass transistor (FZT753SMD recommended). The output current is typically 5.0 mA and it current limits at typically 20 mA. Through the control of the pass transistor, a 5.0 V system voltage is maintained.   This 5.0 V output must provide enough current for the internal components of the ECU and allow for some additional current for external components such as sensors. The 5.0 V supply must also provide high tolerance so that it can be used as a reference for analog signal measurements.

10.4.2.2.2 Implementation Recommendations

The recommended PNP pass transistor has a minimum beta of 50 at 2.0 amps and could easily provide up to 2.0 amps output if it were not for the maximum power dissipation rating of the package which limits it to 2.0 Watts. This can be optimized for an application by taking into account temperature and voltage requirements and selecting a pass transistor with appropriate beta and power dissipation specifications. With a typical battery input voltage of 14 V, the maximum current the voltage regulator can supply using the FZT753SMD, and still be within the 2 Watt rating of the transistor, is 222 mA. If more current were needed, a larger package PNP transistor could be employed. Additionally protection of the 5.0 V can be implemented using a current regulating circuit instead of a single PNP transistor. While this adds components, it allows conditions such as short to ground to reduce the impact of the fault condition. REVO hardware for the reference design uses the FZT789A, which has a higher beta of 175 at 2.0 amps. Future revisions may make use of the lower gain of the FZT753SMD for lower sensitivity.

The VCCSENS pin is the feedback input for the voltage regulator. It ensures that the output of the pass transistor remains at a constant 5 Volts even though the load current required is changing. The VCCSENSE pin requires a capacitor to ground to maintain stability of the regulator. The minimum recommended value of this capacitor is 2.2 μF. but the user can choose to increase this value depending on the ripple voltage other requirements, such as loss of power hold time. The MC33812 requires a reverse battery and transient voltage protected supply. A simple silicon diode in series with the VPWR pin can protect the circuit against an inadvertent reverse connected battery. For transient voltage protection, a TVS (transient voltage suppressor) diode from the VPWR pin to ground is recommended. Also the

VPWR pin should be bypassed to ground with a.1.0 µF. capacitor. These recommendations are implemented and are demonstrated in the schematic.

As a system reference, the control capability of the MC33812 maintains a 2% tolerance over voltage and temperature. From an analog signal measurement perspective, this reference tolerance carries through to the actual measurement result along with other system specific measurement uncertainty. For successive approximation result (SAR) type of analog to digital converters (ADCs) found on most MCUs, this reference voltage dominates the error associated with a measurement. The 2% tolerance from the 5.0 V pre-regulator of the MC33812 provides good performance for analog signal measurement. Ultimately, the end performance requirements for the system must use the analysis of all measurement error components. If the performance requirements for system require additional precision from analog voltage reference, a separate voltage reference is recommended. However, the use of the S12 and S12X MCUs allow extremely high precision measurements to be made without a high precision voltage reference. This is accomplished by using the internal bandgap reference of the MCU. By measuring the internal bandgap, compensation for lower precision analog voltage references can be performed with little overhead. The end result is a significant improvement in measurement performance with no system cost impact.

10.4.2.3 Injector Driver

10.4.2.3.1 Design Criteria

The injector driver is a low side driver with a typical 200 mOhm, RDSON. It is capable of driving most fuel injectors that draw less than 3.0 Amps and must be protected against inductive transients. Additionally, faults associated with conditions that render the injector inoperable, must be detected to prevent system damage and provide diagnostics for repair.

10.4.2.3.2 Implementation Recommendations

Direct control of an injector is performed on the INJOUT pin of the MC33812. The input pin, INJIN, is a logic level (5.0 Volts) CMOS input which can be driven by any GPIO pin from the MCU. However, the control of the pin is typically done through a timer channel configured for output compare. Alternatively, it can be connected to an eTPU channel on an MCU containing this peripheral.

The minimum specified current limit for this driver is 3.0 A. A built-in clamp circuit limits the injector's inductive flyback voltage on the pin to 53 V, typically. The injector driver is an inverting logic element so that when the INJIN pin is high, the INJOUT is low, turning on the injector and vice versa. The injector driver monitors the injector for fault conditions such as shorted coil (short to battery), and open coil or other open circuit connection (wiring or connector). The injector driver circuit protects itself against over voltage, over current, and over temperature. If any of these conditions are present, it indicates the “fault” to the MCU by bringing the normally low INJFLT line, high. The INJFLT line will stay high until the INJIN line goes low and then high again, if the fault has cleared. Detection of the fault is done through GPIO of the MCU. Proper detection of a fault should use interrupts that are triggered when the fault pin goes high. This is implemented on the ECU by connecting all fault pins to Port A pins which have a dedicated interrupt. A simple interrupt service routine can be used to determine which fault pin triggered the interrupt and then determine the type of fault by reading the state of the control pin associated with the load that created the fault. If the control pin is active or in the high state, a short to battery condition can be reported to the main application. If the control pin is in active or in a low state, an open load condition can be reported. Additional detection of over temperature and over voltage can be performed with additional detection code.

Since the INJOUT pin goes to wiring off the circuit board, it must be protected against ESD transients by means of a capacitor. In the reference design, a 10 nF capacitor with a 100 V rating is used on all outputs that go to the P1 and P2 connectors. This is a generic implementation for the defense of ESD type of transients. Specific application requirements should account for the voltage levels and injection methods that come from the operating environment. No component can guarantee meeting a specification. It takes a system level implementation with proper coordination of circuit design, component selection, and layout techniques. This is demonstrated in the reference design by using devices that use the human body model for ESD ratings of +/-2 kV and having ESD capacitors placed at the connector pins placed as stubs between the signal and the connector.

10.4.2.4 Relay Driver

The relay driver is an exact twin of the injector driver, and could be used to drive a second injector, in a two cylinder application. The output pin for the relay driver is ROUT and the input is RIN. There is also a fault indicator pin called RELFLT and it behaves the same as the INJFLT pin. Refer to the Injector Driver section for further details.

10.4.2.5 Lamp Driver

10.4.2.5.1 Design Criteria

The lamp driver specifically designed to drive a Malfunction Indicator Lamp (MIL) on a motorcycle or motor scooter, but could be used for other purposes as well. A MIL is typically an incandescent type of light but may be LED. Operation of both types must be provided. It is a general purpose light indicating the health of the engine and does not require any specific timing requirements. Diagnostic of the light itself is provide on power up and does not require further fault detection of the load itself. A 7.0 W lamp represents a larger type of load used for a MIL.

10.4.2.5.2 Implementation Recommendations

Integrated into the MC33812 is a low side driver for lamps. It is similar to the injector and relay drivers but with less current handling capability. It is designed to have an RDSON of 1.0 Ohm, typically, and can provide up to 1.0 Amp before current limiting. Like the injector and relay drivers, it is protected against fault conditions, however there is no fault indicator pin associated with this driver. It also has over current timer to allow for incandescent bulb inrush currents. The output pin is called LAMPOUT and the input pin is called LAMPIN. The lamp driver can also be used to drive a LED because it does not have a fault detection current sink on the output which would cause ghost lighting in a LED through leakage current. Implementing an LED requires proper series resistance for biasing. Provision for PWM dimming is provided in the reference design by connection of the LAMPIN pin to a PWM output channel of the MCU. This allows fine tuning of the LED brightness and possibility for ambient light compensation. The lamp driver also has a voltage clamp of 53 Volts, typically, so it can drive a relay as well as a light.

10.4.2.6 Ignition Pre-Driver

10.4.2.6.1 Design Criteria

The ignition pre-driver is used to drive an external transistor which controls current flow in an inductive ignition coil. Spark events are then generated by the control of current in the coil. Transistors used to drive the coil can be either Darlington or Insulated Gate Bipolar Transistor (IGBT) type. Circuit must provide system health feedback of the ignition coil drive circuit through the detection of faults.

10.4.2.6.2 Implementation Recommendations

The MC33812 ignition pre-driver has two outputs and one input. The IGNOUTH provides the high side output stage and the IGNOUTL provides the low side output stage. Having both high and low side drive for the pre-driver provides better response for turning on and off current flow in the ignition coil. The MC33812 can be configured to drive either of the two most popular choices for the external transistor: the IGBT or the Darlington transistor. Historically, the Darlington transistor was the most popular choice for the ignition driver transistor. The newer solution is the IGBT, but until recently, it was considerably more expensive than the Darlington. The choice now depends on what the designer prefers and what it more readily available. The reference design uses an IGBT to control the ignition circuit with no provision to configure for Darlington. Further design considerations are included in the Discrete IGBT Driver section below.

For the MC33812, the IGNIN signal is the logic level input that is controlled by the MCU. There is also an input pin to select either the 5 Volt supply or the 12 Volt supply as the source voltage supply for the ignition pre-driver. This input is the IGNSUP pin. If the designer chooses to use an IGBT, then IGNSUP must be connected to 12 Volts to provide the necessary gate drive voltage needed to enhance the gate of the IGBT. When using a Darlington, the IGNSUP pin must be connected to the 5 Volt supply to avoid unnecessary power dissipation in the MC33812 when providing the 50 mA of base current drive. The logic function through the pre-driver is non-inverting, meaning that a high logic level on IGNIN will turn on the IGNOUTH output stage and a low on IGNIN will turn on the IGNOUTL stage. The addition of the driver transistor makes the overall logic function inverting. The IGNFB pin is an input that is tied to the collector of the IGBT or Darlington through a 10:1 resistor divider. The IGNFB is used to monitor the voltage on the collector to check for a shorted ignition coil, or other short to battery conditions. The resistor divider is needed because the voltage on the collector can reach 400 Volts due to inductive flyback from the ignition coil. The divider keeps this flyback voltage down to 40 Volts to protect the input of the MC33812 from over stress. Components of the feedback circuit should be selected to work with the components of the ignitions system. If the IGBT breakdown voltage is different from 400 V, changing the resistor divider is necessary to create the proper ratio that protects the IGNFB pin and create thresholds that properly indicate faults.

10.4.2.7 ISO-9141 Transceiver

10.4.2.7.1 Design Criteria

For small engines, the number of electronic modules is typically very limited. In most cases, the engine controller may be the only ECU. While communication to other ECUs is not apart of normal operation, it is typically required to provide diagnostic communication to troubleshoot the system. The ISO-9141, also known as a “K-Line” interface, allows bi-directional serial communications between the MCU and an external master device. It is typically used to convey diagnostic messages between MCU and an external diagnostic code reader. A common small engine implementation will use ISO-9141 specifications as the physical transport layer for communication between the ECU and the diagnostic tool. ISO-9141 is a good choice as it is a single wire interface and protocols are based on a UART functionality commonly found on most MCUs. Connectivity is also robust as it uses battery level signals.

10.4.2.7.2 Implementation Recommendations

Integrated into the MC33812 is an ISO-9141 transceiver. The MTX and MRX pins are the logic level input and output pins, respectively, that connect this block to the SCI port of the MCU. The ISO-9141 pin is a 0 to battery voltage interface pin with an active pull down MOSFET and a passive pull-up resistor of 1.0 kOhm. In the reference design, this I/O is also protected from reverse battery by a diode and from transients with a capacitor and a 24 Volt TVS.

Beyond the ISO-9141 communication, provision for adding connectivity to a CAN transceiver has been provided on the reference design. The 4-pin header label “CAN1” provides power and ground connections as well as CANTX and CANRX connections back to the MCU. This allows future expansion and connectivity to a CAN bus containing multiple ECUs. The use of CAN in a small engine system will become more important as electronic content increases.

10.4.2.8 Power On Reset Generator and Watchdog

10.4.2.8.1 Design Criteria

During conditions such as power on or transients in battery voltage due to high load or low battery, it is required that the system behave in a deterministic and controlled manor to prevent damage to the engine or ECU itself. This results in the demand for an external monitor of the system voltage. The circuit must have the ability to shutdown or prevent operation of the MCU when the power gets near the limits of normal operation. The most common implementation for this is for an external device to hold the RESET pin of the MCU low unless the 5.0 V system power is within the limits of normal operation. This aids in preventing situations leading to code runaway. Additional safety is sometimes required to ensure that the application running on the MCU is not lost or executing in an unexpected way. An external watchdog circuit requires the MCU to periodically communicate to an external chip to indicate that software has not slowed execution or runaway. If the MCU fails to provide the proper feedback to the external watchdog, it will toggle the RESET pin of the MCU and bring the software to a known state.

10.4.2.8.2 Implementation Recommendations

If the MCU needs an external voltage monitor to provide a reset signal, when the supply voltage is first applied, or it goes out of range, the MC33812 provides this function. Most MCU's must not be allowed to begin processing until their 5 Volt supply is stable and within the required range. The POR (Power On Reset) circuit provides this function. When power is first applied to the MC33812, the RESETB pin is held low by an internal pull down resistor. When the POR circuit sees the VCCSENSE pin exceed ~4.75 Volts, it starts a 128 μs timer. When this timer, times out, the POR circuit raises the RESETB pin which allows the MCU to come out of reset and start processing. This is one of the functions of the POR circuit. Another function of this circuit is to continuously monitor the VCCSENSE pin for any voltage transients that could cause the VCCSENSE voltage to go out of range. These voltage interruptions could cause the MCU to lock-up or begin executing erroneous program instructions. When any out-of-range voltage conditions are detected, the reset generator will bring the RESETB pin low for 128 μs and, once the condition has cleared, will bring the RESETB pin high again to allow the MCU to restart.

The last function of the reset generator circuit on the MC33812 is the watchdog. The watchdog is a timer circuit that can be programmed with a specific time value, between 1.0 ms and 10 seconds. Once the watchdog timer is programmed, the MCU must send it a pulse on the WDRFSH pin, at least once during the programmed time period, to avoid the watchdog from issuing a reset to the MCU. The purpose of the watchdog is to monitor the MCU to ensure that the program code is running and that the MCU has not “locked-up”. If the MCU enters an infinite program loop or it executes an erroneous halt instruction, the watchdog of the MC3812 can toggle the RESETB pin. The MCU must be programmed to “pet” the watchdog, (i.e. send it a pulse on the WDRFSH pin), periodically, to keep the watchdog from issuing the reset to the MCU. Care must be taken so that the “pet” of the watchdog is done as a natural part of the main loop execution and not down by means of a hardware timer.

To program the watchdog timer initially, the MCU must send it a pulse greater than 1 ms but, less than 10 seconds on the WDRFSH line. The default value for the watchdog timer is 10 seconds. If the MCU does not initialize the watchdog timer, the default value will remain in effect and the watchdog will issue a reset to the MCU on the RESETB pin when it times out, in 10 seconds. Whenever the watchdog issues a reset, it reloads the default time value, 10 seconds, into its timer. If the watchdog is not needed, it can be disabled by pulling the WD_INH pin high through a pull-up resistor to 5.0 Volts. For more details on setting and using the watchdog timer please see the MC33812 data sheet.

On the reference design, the RESETB pin is connected to the RESET pin of the MC9S12P128 MCU through a 2.2 kohm resistor. This provision is made so that an external tool can be connected to the BDM programming header and overdrives the RESETB pin of the MC33812. By implementing this, an external tool can maintain control of the RESET pin of the MCU during the programming sequence and prevent an erroneous reset during programming.

Additionally, a small capacitor is placed near the RESET pin of the MCU. This is used as a precaution to filter noise that can lead to a glitch on the RESET pin. Typically this is not populated in production but is available as a placeholder that can be populated as electro-magnetic conformance or radiated immunity testing demands. Sizing of this capacitor should be balanced with the RESET pin functionality of the MCU. Specifically, any internal reset of the S12P will drive the pin low for 512 PLL Clock cycles and release. The S12P will then sample the RESET pin 256 cycles after the release to determine if the reset event was internal or external. Sizing this external capacitor must take this into consideration along with the any external pull up current source. For the reference design a 100pF capacitor was selected with the 10K pull up resistor. This is based on a worst case maximum PLL Clock speed of 32MHz and obtaining a valid high signal level after two time constants before the 256 cycles after the RESET pin is released. Further details are provided in the datasheet for the S12P device.

An alternative solution for connecting the RESETB pin of the MC33812 to the MC9S12P128 MCU is the XIRQ pin. By connecting the RESETB pin to an unmaskable interrupt source, the MC9S12P128 can use its own reset detection circuitry and watchdog. This type of configuration would allow the MC9S12P128 to operate down to its minimum voltage of 3.13 V. Refer to datasheet of MC9S12P128 for further details. The advantage of running down to this voltage level is increased time from a loss of power event to loss of operation. Providing the maximum time during a loss of power event allows the size of the bulk storage capacitor on the 5.0 V system voltage to be reduced in size when compared to 5.0 V only operation. The total time required between a loss of power and loss of operation is heavily driven by the amount of data necessary to be stored in non-volatile memory during system shutdown. If the system can perform to a lower voltage, it has a longer time available for operation on a given capacitance. The design can then optimize the bulk charge storage on the system supply to incorporate a small capacitor, which can provide lower cost.

10.4.3 Power Supply

10.4.3.1 Design Criteria

System power is derived from a 12 V battery. The battery has a nominal 13.8 V output which must be reduced to a system voltage of 5.0 V and provide at least 150mA to the system including external modules. Performance of the 5.0 V supply must have greater than 5% accuracy to prevent the need of a separate voltage reference for analog measurements.

10.4.3.2 Implementation Recommendations

The main control for the power supply is integrated into the MC33812 device through a pre-regulator. This is discussed in detail in the 5.0 Volt regulator pre-driver section for the Small Engine IC block.

While the reference design implements a hard power on strategy through the ignition switch, it is possible to architect a power control circuit that enables a more controlled power down process. This would enable the MCU to control when power down occurs and reduce the constraints associated with power off detection. For such a system, the module would incorporate an ignition switch signal, and a direct connection to the battery. Power coming from the battery would go to control logic that performs an OR logic function between the ignition switch signal and a signal from the MCU. This allows the battery power to turn on when the ignition switch is keyed on and when the MCU has determined it needs power. When the key is turned off, the MCU can hold power on through the OR logic and conclude any necessary activity, such as storage of data to non-volatile memory, and then shutdown power. This functionality is very attractive but has its trade-offs. The main trade off is cost. Additional components are required. The second is reduced operating range. Since additional components are placed in the battery power path, additional tolerance and error is presented to the battery voltage powering circuits, such as the 5.0 V regulator. Low battery conditions suffer the most as the additional drop across the control logic reduces internal voltages, creating a smaller operating range.

10.4.4 VRS Conditioning

10.4.4.1 Design Criteria

The variable reluctance sensor (VRS) conditioning circuit is used to convert the analog signal from the crankshaft variable reluctance sensor into a logic level signal that the MCU can measure and extract timing information from. A large AC differential signal from the sensor is conditioned to be a pulse where edges represent the edges of the toothed-wheel on a rotating engine crankshaft. Since the amplitude of the VRS signal ranges from less than 1.0 V over 70 V in most applications, the large gain of the conditioning circuit must also reject noise.

10.4.4.2 Implementation Recommendations

The circuit chosen for the reference design uses the Maxim MAX9924. This device was selected because it contains both a precision differential amplifier and a comparator with selectable adaptive peak threshold and zero-crossing circuit block, all in a small 10 pin µMax package. The circuit is capable of operating in four different modes noted A1, A2, B and C. The data sheet for the MAX9924 explains the pros and cons of each mode and for the reference Design we opted for mode A2 because it was the simplest mode to implement and required the fewest components. Custom implementations may incorporate the external threshold voltage and provide interaction from the MCU based on the operating point of the engine. This would allow the detection voltage of the VRS signal to be changed based on operating conditions. This specifically benefits startup timing precision.

Additionally, the reference design can accommodate the use of a Hall Effect Sensor. Since Hall Effect is essentially a low side switch that grounds a 5V signal on tooth edges, it provides good detection with much less sensitivity than a VRS. To use a Hall Effect Sensor with the reference design, a short must be placed from the VRSP input pin to the VRSOUT signal of the VRS conditioning circuit. This is provided through R15, which is not populated in production. While Hall Effect Sensors provide cleaner signals and simpler connection to the MCU, they potentially add system cost due to the sensor design itself.

10.4.5 Dual H-Bridge

10.4.5.1 Design Criteria

The idle bypass valve directly addresses issues with associated with running a cold engine. This is the replacement for a choke found on carbureted engines and is electronically controlled by the MCU. Two architectures are known for controlling an idle bypass valve. The difference is the type of electric motor that is used to vary the amount of air flow in the bypass valve. A DC motor with a position feedback sensor and a bi-phase stepper motor are the two possible solutions. Stepper motors require more complex control techniques but allow for a simpler mechanical design and less calibration. A small stepper motor used as an idle air speed motor typically has less than 1.0 A peak current and requires a dual H-bridge circuit to control. The stepper motor is also an inductive load and requires suppression of the flyback voltage. Since the idle speed motor provides a key role in the start and idle of an engine, it also plays an important part in the start up and low speed emissions. This then requires diagnostic information about the health of the idle speed motor to maintain emissions performance.

10.4.5.2 Implementation Recommendations

To create the dual H-bridge circuit used to control a stepper motor, the MC33879 is used. Early versions of the reference design may use the pin for pin compatible MC33880. The MC33879 is a configurable output device capable of delivering a minimum of 0.6A for each leg of the dual H-bride. Each output is protected through current limit and inductive clamping and can sense faults including open load and output shorts. Control and feedback of the outputs is performed through a serial peripheral interface (SPI). This common interface is connected to the SPI of the MCU.

As the MC33879 is a general purpose load driver, it is not optimized for stepper motor control. However, simple accommodations in software allow it to work well with a stepper motor. For motor control, dissipation of energy in an active coil is important before changing the direction of the current flow. This is called dead-time insertion. For the implementation of the MC33879 as a stepper motor driver, dead-time insertion must be handled manually through software. The result is the addition of new states to the stepper motor state machine to prevent shoot-through that could lead to a system fault.

The configurable functionality of the MC33879 makes it a good choice for implementations other than a dual H-bridge. Paralleling outputs can create a higher power device that drives loads requiring greater than 0.6 A of current. This makes it possible to drive a DC motor if used for idle speed control or other purposes. Other possibilities for loads include relays and LEDs.

10.4.6 Discrete Low-side Driver

10.4.6.1 Design Criteria

Provision must be made for driving unspecified loads in the system. These loads can be motors, relays, lamps, LEDs, solenoids, or other high current devices. As the load may be an inductive type, flyback voltage must suppressed on the output.

Implementation Recommendations

Two discrete low side drivers are implemented through two dual-channel NMOS transistors. These control the O2HOUT and ROUT2 signals. Each dual-channel transistor is paired together to make a single high power output. Additionally, the output has a 1500 W TVS to suppress transients that may occur on the unspecified load.

This is a generic implementation for a discrete low side driver. Optimization of the circuit begins with sizing the transistor to match a specific system load. The same should be performed with the transient suppressor. Components of the discrete low side driver are quite large and selected based on the high drive current beyond a normal application demand. Specific implementations may also take into account current feedback and fault detection through analog measurement. This combined with over-current protection would create a low side driver that is much more capable for a specific application.

10.4.7 Discrete IGBT Driver

10.4.7.1 Design Criteria

The IGBT must be capable of carrying greater than 2.0 A and have a breakdown of at least 400 V to drive the example inductive ignition system. Packaging must be small and cost effective. Device must maintain minimum operating characteristics over temperature.

Implementation Recommendations

A small TO-252 sized IGBT was selected capable of delivering 10 A with a breakdown voltage of 430 V. The breakdown voltage for an example ignition system was found to be 420 V. This parameter is dependent on the actual system it is implemented on and a device with appropriate performance selected. Power dissipation is a large concern for the IGBT. Worst case operating conditions should be considered for power dissipation to validate the use of the selected component package.

An additional IGBT can be implemented for a two-cylinder application that has two independent ignition coils. This second ignition driver would require the capability to be driven directly from the MCU. This would give up the fault detection on the second coil that is provided by the MC33812. Diagnostics for the second ignition coil could be created through additional circuitry beyond the IGBT. Such a circuit would also require the use of an analog channel of the MCU to detect faults.

10.4.8 Input Signal Conditioning

10.4.8.1 Design Criteria

All sensor and switch inputs must have a filter to reduce noise. The filter must provide appropriate response for each type of signal.

10.4.8.2 Implementation Recommendations

Low pass filters are implemented on each input signal based on a nominal frequency response:

Input Signal Frequency Response

Module Pin	3.0 dB Frequency Cut-off
TPS	1.0 kHz
ATEMP	100 Hz
ETEMP	100 Hz
MAP	5.0 kHz
O2	500 Hz
TILT	100 Hz
ENGSTOP	100 Hz

These filters can be adjusted to obtain the desired response. As a design recommendation for using the analog pins of the MC9S12P128, a minimum capacitance of 10 nF should be maintained at the analog pin. This ensures that transfer of charge during an analog measurement does not impact the measurement result. Placement of this capacitor should be near the analog pin to reduce noise and minimize impedance.

Digital inputs associated with switches must be designed for the desired voltage range. This is done by altering the voltage divider between the ECU pin and the MCU pin. Switch inputs for TILT and Engine Stop are configured for 12 V operation and may require additional validation based on actual voltages.

The battery voltage is a special case input. The battery input is subject to intensive transient conditions and requires additional consideration beyond a simple filter. At the connector input, it must be protected to survive conditions such as reverse polarity and load dump. These are addressed in the reference design by a reverse blocking diode and a 1500 W TVS. This circuit should be modified to address specific application demands. Sizing and response of these components should be optimized to very specific test pulses that represent known transient conditions. For the actual battery voltage measurement itself, a separate circuit should be considered to balance measurement accuracy with transient protection. The reference design uses a simple reverse blocking diode with good tolerance of the forward drop voltage. This is important to maintain the integrity of the battery voltage measurement. Poor tolerance on the input diode leads to poor accuracy of the battery voltage measurement. Alternative battery measuring circuits can provide better protection beyond reverse battery. As these circuits incorporate additional components care must taken to ensure a battery measurement can be made to meet the performance goals of the system.

11 Appendix B: Software Reference Manual

11.1 Software Architecture

The example application software package is designed to abstract the low level details of using a microcontroller in an engine control application. This will allow a developer to focus attention on the signals and application tasks instead of on the very specific functionality of the MCU. Software applications can then be created based on a higher level of understanding. For the Small Engine Reference Design, this high level approach breaks the engine control application into three tasks: 1) User Management, 2) Data Management, and 3) Engine Management. Each of these tasks must be developed by the user. Working examples for running a real engine are provided as model of successful using this architecture.

The hardware abstraction layer uses complier directives to associate software signals with hardware functionality. This is best shown in the following example:

```
/** RIN3, Port P, Channel 2 */
#define     RIN3        (PTP_PTP2)

/** RIN3, Port P, Channel 2, Low */
DDRP_DDRP2 = OUTPUT;
RIN3 = LOW;
```

In this example the relay 3 control signal, RIN3 is being removed from the hardware in two ways. First, the actual control pin PTP2 is associated with the system signal name RIN3. If a change is made, based on a hardware level modification, it can quickly be made at for every instance of the signal by only changing one line of code. Also the control for the signal is referenced by functionality, in this case a low signal. This is important as a signal could have reverse logic due to other components outside the MCU and can be easily modified to accommodate such a situation.

Under the high level functions, are control tasks. All control tasks operate regardless of the completion of the high level tasks. This creates a hybrid operating system where high level tasks are in the time domain and the low level tasks are in the crankshaft or angle domain. Low level tasks in the angle domain operate on an event basis and then get the latest operation parameters from the time domain tasks on the conclusion of an event. As a result, if a time domain task did not complete before the next angle domain event associated with the task, the angle domain event will occur with the previous parameters. This is an acceptable practice as the response time of an engine is proportional to its operating speed. As the engine rotation rate increases, there is less time for the application to execute. However, the inertia of the engine changes with the rotation rate, which makes the engine less sensitive to fine control adjustments as engine speed increases.

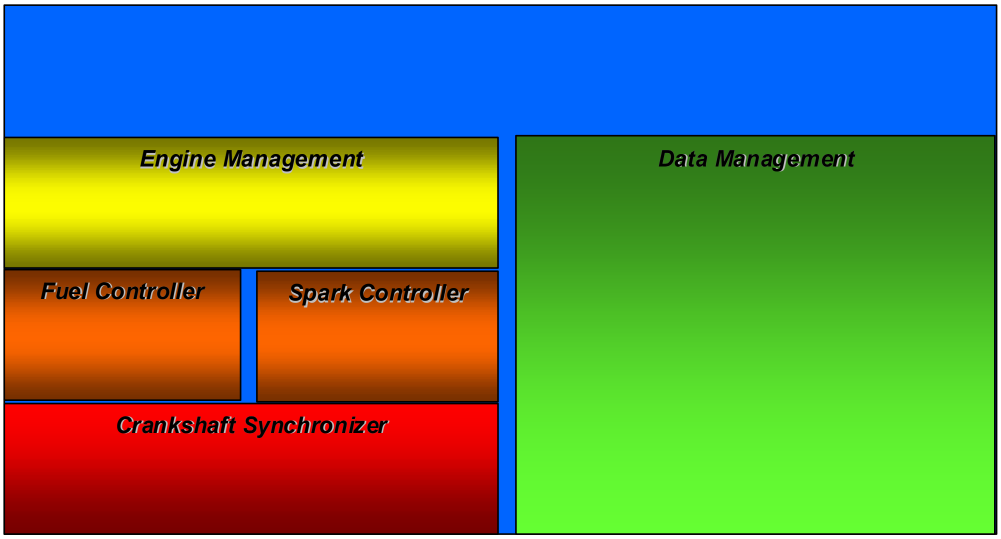


Figure 24. Visual Diagram of the Software Architecture used for the Small Engine Reference Design

At the lowest level is a crankshaft state machine that responds to tooth driven interrupts. The main task of this state machine is to validate each tooth edge and maintain a synchronized state that can create angle based events. Activities include synchronizing to the missing tooth gap, validating a tooth period, counting teeth, scheduling future fuel and spark events.

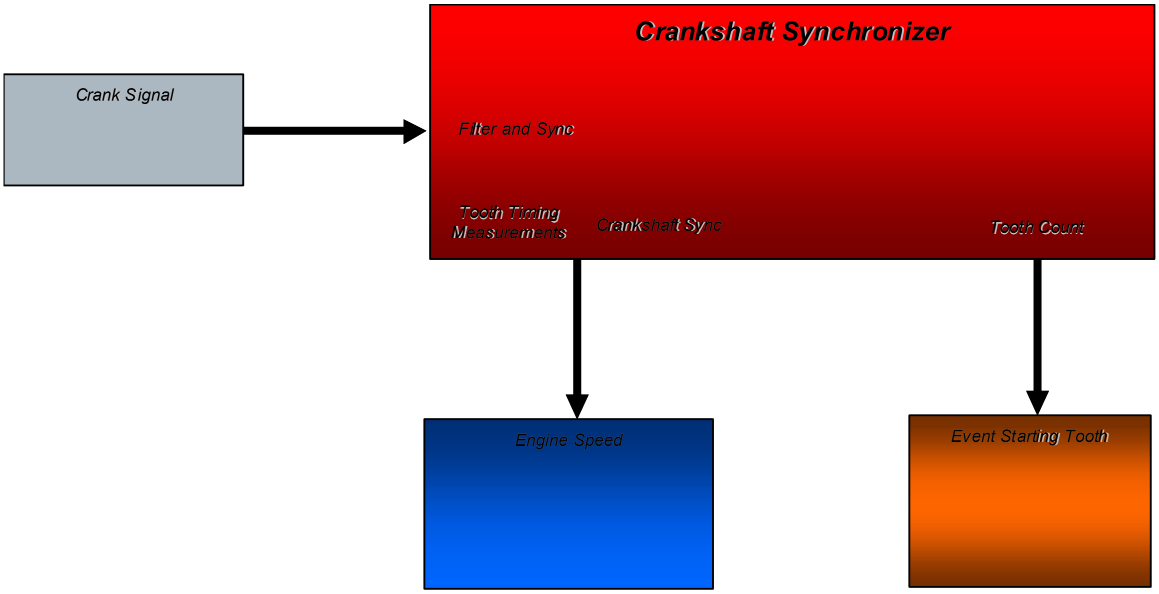


Figure 25. Crankshaft Layer Data Flow

Once the crankshaft state machine reaches a synchronized state, it can then begin scheduling events for the fuel and spark controllers. These events are based on the single action timer channel hardware of the MCU. For this type of hardware capabilities, the best precision possible is to use the tooth interrupt as a known point in the rotation of the engine. Using the time between the last two teeth and the angle this represents, the angular rate of rotation can be calculated and then used to schedule the start of an event in the future. When the start of the event occurs, the end of the event can be scheduled based on the current data parameters of the fuel and spark controllers.

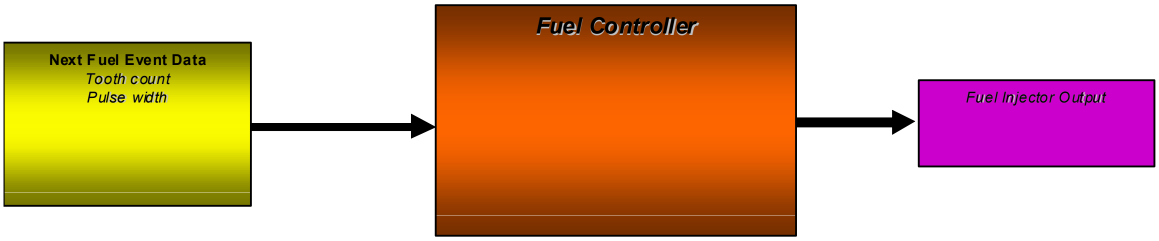


Figure 26. Fuel Controller Software Model

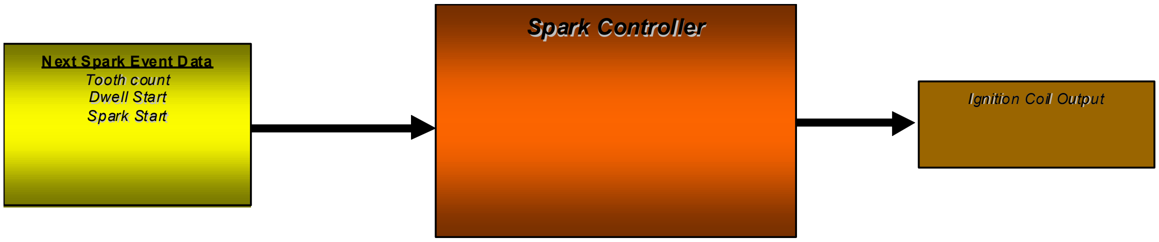


Figure 27. Spark Controller Software Model

New data related to when the fuel and spark events occur and for how long, are determined by the Engine Management task. The basis of this new data is a table look up using engine speed and load data provided by the User Management task. Once the table look up takes place, any modifiers to the base table look up can then be added to create a value that can be used by the fuel or spark controller. To prevent complications and undesired operation, any new data calculated by Engine Management will go into a variable that will not be loaded by the fuel or spark controller until the current event completes. This lockout mechanism prevents malicious modification to fuel or spark timing during a fuel or spark event.

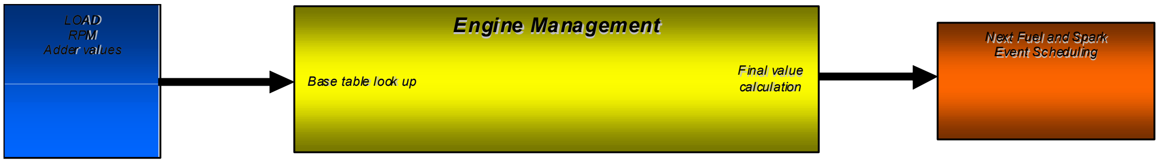


Figure 28. Engine Management Software Model

All data collection is performed by the Data Management task, with two exceptions: engine speed and optionally MAP data. As the crankshaft controller uses tooth period data, it makes sense for it to collect this data for use. Measuring MAP at specific teeth has strong benefits to the system and allows four-cycle synchronization without a cam sensor. The Data Management task is designed to collect data at a periodic rate and fill data buffers. Once the data buffers are full, the user can then run filter algorithms to provide User Management with conditioned data.

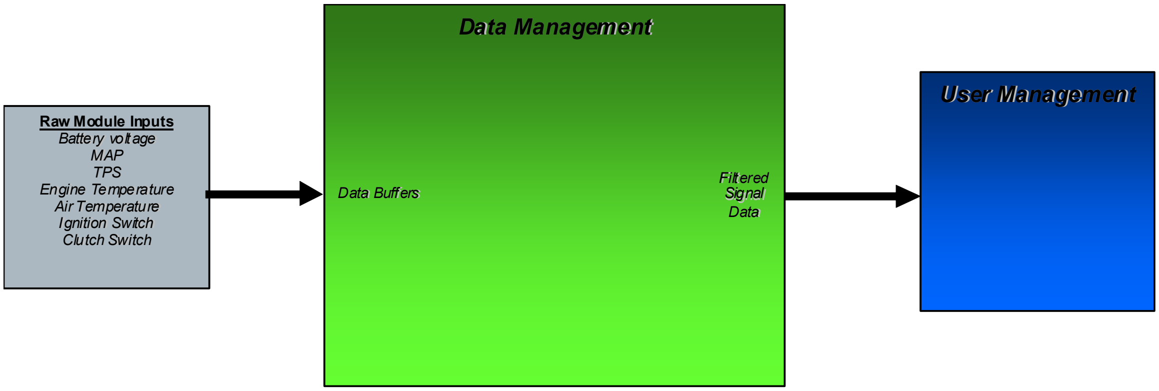


Figure 29. Data Management Software Model

The main system control is performed in the User Management task. All engine control strategy and operator interface control is in this function. A basic state machine is included in the new project as well as the demo application. This application state machine is based on engine speed and is divided into 5 engine conditions: INIT, STOP, START, RUN, OVERRUN. Each of these states represents a typical operating condition an engine user/operator would like the application to manage. The conditions for going between the states and how it impacts the lower level controllers is the heart of the application that is to be created.

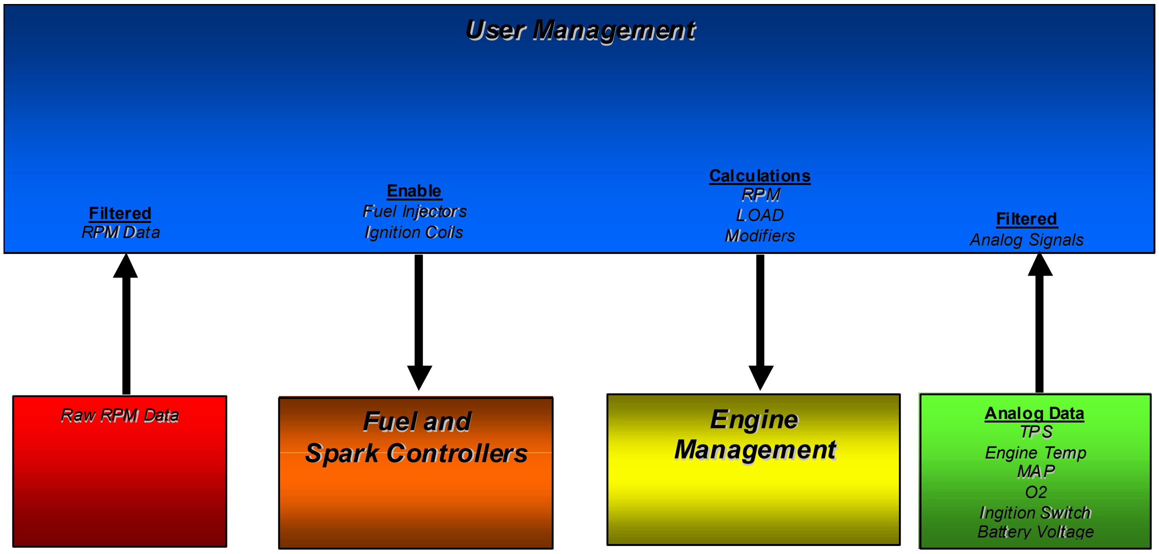


Figure 30. User Management Software Model

11.2 Building Application

Creating a new engine control strategy takes significant experience and knowledge. It is recommended that you work with one of the demo applications to get a feel for a basic application using this software before creating a new one. If you choose to work with a ground up application or a demo one, there are several files that are intended for use: User Management.c, Engine Management.c, and Data Management.c. Each file also has an associated header file. Additionally, each file is designed to contain a single task that is periodically executed. These tasks are User Management(), Engine Management(), and Data Management(). The file Tasks.h is used to determine when each of these application tasks are called and should be modified by the user to customize the application. Tasks are broken out into 1.0 ms, 2.0 ms, 10 ms, 50 ms, and 100 ms function calls. The scheduler is very simple and the user must take into account the real time aspects of making these function calls.

11.2.1 User Files

The software architecture allows the user to work with four C-code files to create a custom engine control application, User Management, Data Management, Engine Management, and Application Maps. Each of these files has a header file associated with it as well for declaring functions and making definitions specific to the application. Each of these files, with the exception of the Applications Maps, has a specific function call associated with it that is invoked by the task scheduler. The scheduling of these tasks can be modified in the tasks.h file as discussed previously. The tasks.h file should include a function call to each of the user functions: User_Management(), Data_Management(), and Engine_Management(). These user functions provide interaction with the low level driver functions necessary to control an engine.

11.2.2 User_Management()

This function provides the interface between the engine user/operator and the engine control software. The highest level of control is performed through this function. It must take the user inputs and turn them into data for engine control as well as maintain basic operating conditions related to engine's current operating point. For example, a fundamental user control is the throttle. User_Management() will take the throttle position data and provide this data to the fuel and spark controllers so that the fuel and spark is adjusted for the current throttle position. Another fundamental user control is the engine kill switch. The ability of the user to shut off the engine at anytime must be maintained through the User_Management() function.

Examples of a more complete User_Management() function is provided in the example application. This examples use a state machine approach as running an engine has natural control states desired by most users. When getting started, it may beneficial to keep it simple and have a minimal number of inputs. This will allow the designer to get familiar with the entire software architecture before getting into a more complex application.

11.2.3 Data_Management()

All inputs to the system are collected by the Data_Management() function. There are two exceptions. Engine tooth/position data is handled specifically by the low level crank position functions. Also, the MAP signal may be defined to be collected on a tooth basis, not a time basis, which is defined in the application header file.

Digital and analog signals are collected and filtered by the Data_Management() task. Each signal must be defined and configured in the application header file. The Data_Management() must then collect the data and filter the data when buffers are full. The filtered data can then be used by User_Management().

11.2.4 Engine_Management()

The interface to the spark and fuel controllers is through the Engine_Management() task. Using the RPM and load data provided by the User_Management() and crank position function, the Engine_Management() task makes function calls which perform a table look up of fuel and spark parameters based on the current operating point. These parameters can then be compensated based on other modifiers determined by the User_Management() and then sent to the fuel and spark controllers for use on the next fuel or spark event.

A lock out mechanism exists between the fuel and spark controllers and the Engine_Management() function to prevent undesired operation. The fuel and spark controllers create events based on a current variable. Engine_Management() can only modify a next variable. When a fuel or spark event completes, the current variable is updated with the next variable.

11.2.5 Application Maps

The table look up performed by the Engine_Management() task works with the Application Map source and header files. These two files create the fuel and spark tables used to control the engine. The header file is used to configure the size of the tables. The C source file contains the actual data for the table look up procedure. In addition to the table parameters, the values associated with the table indices must also be created. This includes a data array of the RPM values and an array of the load values that correspond to the indices of the fuel and spark tables. All data in the application maps is based on microcontroller timer units, not engineering units. This is noted in the example files. As a fundamental exercise in running an engine, these tables must be created based on the engine and system design. A starting point can be obtained by collecting data from an existing engine controller running the same engine or by engine modeling software capable of creating a baseline volumetric efficiency map.

11.2.6 Low Level Driver Files

The software architecture for the example application uses a hardware abstraction layer that removes details of working with the S12P microcontroller. As a result, exercising the signals of the ECU do not require specific references to MCU signals or peripherals. However, it is worth noting that there are limitations and simple tasks at an application level may have significant overhead associated with them. For example, the S12 is a 16 bit microcontroller that does not have a floating point unit. Use of 32 bit data and floating point numbers should be extremely limited and is not recommended for highest performance. Also it is important to note that the reference design is a system and modifying engine control signals may require interaction with another integrated circuit. This type of interaction and system architecture results in many low level software drivers that are behind the scenes. These drivers provide high level functionality for the application and are a key to rapid application development. While it is not vital to knowing all the low level functions, if will be important during the debugging phase that they exist as stepping through them will occur.

12 Appendix C: Calibration

Reserved for future use.

13 Appendix D: References

AN3768, http://www.freescale.com/files/microcontrollers/doc/app_note/AN3768.pdf?fsrch=1, July 7, 2009, Toothgen, page 1

Crankshaft Timing Signal Wheel Simulator, http://www.megamanual.com/router/crankwheelpulser.htm, July 7,2009