Appendix A. Software

Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation

January 18, 2011

REDACTION NOTE

Since public release of this appendix on February 8, 2011, the Agency has revised its redactions to the document to release certain material previously deemed confidential under U.S.C. § 30167. This document, which was posted April 15, 2011 to NHTSA's web site, replaces the one posted previously and contains the Agency's revised redactions.

Table of Contents

List of Figures

List of Tables

A.0 Software Analysis

This appendix provides a brief introduction to the context of the team's study into possible software causes for unintended acceleration (UA) in Toyota vehicles.

A.1 Organization

The software study was one of several studies performed by the NESC under the direction of the National Highway and Transportation Authority (NHTSA) and the Department of Transportation (DOT). The software study concentrated on the potential triggers that could be hiding in the software.

A.2 Scope

The study focused on the 2005 Toyota Camry L4 (inline four cylinder) engine control module (ECM) software. The selection of the make, model, year, and configuration was determined by the NASA. As noted in the DOT Test Plan:

> "The area of emphasis will be the 2005 Toyota Camry because this vehicle has a consistently high rate of reported 'UA events' over all Toyota models and all years, when normalized to the number of each model and year, according to NHTSA data."

Toyota provided several different versions of the ECM software including versions for 2007 and V6 engines; however, given the limited amount of available time, the study focused primarily on the 2005 Toyota Camry L4 (inline four cylinder) engine control module (ECM) software.

The study started mid April 2010 and ran through mid August 2010. Initially, the software study was supported by the Toyota facility in Torrance, California. The effort expanded to two facilities; one in Torrance and a second facility at the legal offices of Bowman and Brooke's Law Office in San Jose, California. This second facility became operational in June 2010.

A.3 Facilities & Resources

In order to support the confidentiality of the 2005 Camry source code, the two facilities were maintained within offices controlled by Toyota. The software teams traveled to these offices from Ames and JPL for study of the source code and Toyota documentation.

The Torrance facility was located on the 7th floor of a Toyota building located near the main campus of the Toyota US headquarters. The San Jose facility was located within the Toyota supported legal offices, and had access to the conference rooms.

As noted, the NASA engineers performed the study on Toyota premises within an access controlled area. The NASA Software team/NHTSA/DOT had agreed not to remove Toyota intellectual property from this location, most notably software source code and design documents. Toyota provided all computational resources for this work, to the team's specification. For example, the resources that were provided at the Torrance facility Toyota provided four desktop workstations, in dual-boot configuration supporting the Windows XP and the Ubuntu/Linux operating systems, a backup system, and a large compute server with 32 GBytes of main memory and 12 CPU cores, for the computationally more intensive model analyses. Access to the Toyota source code was made possible through the workstations. A similar setup was provided by Toyota at the San Jose facility, except for the multi-core system.

The team was able to start work within a week of the initial proposal. Throughout this study, Toyota IT personnel were also made available to assist with any technical problems with the workstation setup.

Software tools provided by Toyota included the Atlas translation software system for rough online translations from Japanese into English, and a version of the compiler suite that Toyota uses to compile their source code. The proximity of the work area to Toyota headquarters facilitated a direct interaction between the NASA software team and the Toyota engineers. The discussions took place in English and Japanese, with the help of an interpreter who provided two-way translations during all regularly scheduled and impromptu meetings and discussions.

During the study, JPL, Caltech, and Ames intellectual property employed to analyze the Toyota software was protected. This included encrypting tools and the data archives with the intermediate results. This step was warranted given that the analysis was performed on Toyota premises using Toyota computational resources.

The nominal work day started at 8am and concluded at 7 pm. Throughout the study, four regularly scheduled daily meetings were held with Toyota engineers. The meetings served to structure the team's interactions with Toyota and facilitated the process of obtaining technical details where needed, on an ongoing basis.

- A first meeting was held at 9am each day to summarize the day's plans for both sides.
- At 1 pm an 'action item debrief' meeting followed, to cover any action items provided by the team to Toyota that were ready to be closed with the delivery, by Toyota, of the corresponding response or documents. This debriefing took from one to three hours.
- At 4 pm a 'new action item' meeting was held wherein the NASA team could submit new actions to Toyota.
- Finally at approximately 6pm, the Toyota engineers in Torrance held a teleconference with Toyota in Japan to relay the team's new action items to Japan. Because of the time

difference to Japan (+16 hours), this teleconference allowed the Japan-based engineers to work through the (Pacific Time Zone) night to address the team's action items.

Based on the team's requests for technical data, Toyota flew domain experts on various topics to the Torrance and the San Jose offices to address the team's questions in person and to respond to follow-up questions. Most of the Toyota experts stayed for one to two weeks, before being replaced by a new team. In this way, the NASA team was able to have detailed interactions with, among others, Toyota throttle control system and software engineers, cruise control system engineers, transmission engineers, and brake system engineers. Additionally, Toyota management representative from Japan were also on site to organize the responses. Both Toyota engineers and DENSO engineers (the software contracting company employed by Toyota in the software development), were made available for this study. The US Toyota president and COO of Toyota, Jim Lentz, also visited the NASA team during this study on May 13.

Toyota attempted to provide information as quickly as possible, including document translations. Questions about the provided information could always be resolved in follow-up conversations or by referring to documentation that was also provided. Most questions were asked several times during the study to assure the teams understanding.

As part of the study, the software team generated over 100 separate action items. Each action item was discussed in the face to face meetings, as were the Toyota responses. All documentation responding to the action items was provided by Toyota on a secure file server on-site. The server also held copies of the Camry software source code (for multiple Camry years and versions), the design documentation, and a limited number of unit-test case source files. While on-site in Torrance and San Jose, access this server allowed for the review and use of all information provided.

A.4 Technologies Applied

A.4.1 Software Implementation Analysis Using Static Source Code Tools

The initial focus in analyzing the Camry05 source code has been on a thorough static source code analysis of the ECM to find possible coding defects and potential vulnerabilities in the code. Toyota internally makes use of the QAC tool for static analysis. The team's experience is that there is no single analysis technique today that can reliably intercept all vulnerabilities, but that it is strongly recommended to deploy a range of different leading tools. Each tool used can excel at a different aspect of static analysis, which results in remarkably little overlap in the set of warnings that is produced. The combination of different analyzers achieves the highest value in analysis results. Three different static source code analyzers were deployed in for the study, which made use of the respectable static analysis capability in modern compilers, such as the public domain standard gcc compiler suite, version 4.

Tools Used:

- Coverity[1] – is currently one of the leading static source code analysis tools on the market. It excels at finding common coding defects and suspicious coding patterns in large code bases, taking a relatively short amount of time. The tool also supports custom-written checkers that can verify compliance with user-defined additional coding rules. This capability is put to use by using a small set of such Extend checkers. Coverity aims to reduce the number of warnings it issues to a minimum, by a careful filtering process that seeks to identify the most relevant or critical issues.
- CodeSonar[2] – is a second strong static source code analysis tool from Grammatech that uses a different technology for detailed inter-procedural source code analysis. Codesonar analysis typically takes longer to complete than comparable tools, but can reveal more subtle types of defects and suspect coding patterns, requiring deeper path analysis (which can be more time consuming). The version of Codesonar used was extended with checkers for JPL's coding standard. In this study the software team separated results for the coding standard checks from the default results.
- Uno[3] - is a research tool for performing static source code analysis, originating at Bell Labs. It is designed to perform a simpler, fast analysis for intercepting primarily the three most common types of software defects in programs: the ***U***se of uninitialized variables, ***N***il-pointer dereferences, and out-of-bounds array indexing. The tool can be extended with user-defined checks.

A.4.2 Software Logic Model Checking

Logic model checkers use efficient algorithms to explore all possible executions of a system in an attempt to locate those executions that violate user-defined logic properties. The exploration is in principle exhaustive and thus if it completes without finding incorrect behavior then the system is proven correct. Because the exploration is exhaustive, model checkers can excel in finding incorrect behavior in 'corner cases' which are exactly the types of errors that tend to occur infrequently and be overlooked in even rigorous standard software testing. Key to a logic model is the formulation of the model's behavior and its correctness.

[1] http://coverity.com/

[2] http://grammatech.com/products/codesonar/overview.html

[3] http://spinroot.com/uno/

The tools used in the for logic model checking were:

- Spin - is an open-source software tool for the formal verification of distributed software systems. In April 2002 the tool was awarded the ACM System Software Award for 2001. It is possible to use this tool both for the exhaustive verification of high level design models of a system and for the detailed exploration of implementation level code in multi-tasking or multi-threaded systems.
- Swarm – is a preprocessing system for Spin that can maximize use of available compute resources in large compute clouds or grids thereby allowing for a comprehensive analysis of large and complex software systems.

The logic models developed focused on those modules within the Toyota throttle control software that were deemed, based on the team's study, most likely to contribute to unintended acceleration. The modules selected are related to:

- the conversion of the desired throttle position into a pulse width modulated set of H-Bridge transistors signals;
- the generic processing of the sensors inputs of any type; and
- the specific processing of the accelerator pedal position sensor input.

This set of logic models are involved with the input/output of the throttle control software. Possible errors in modules related to the output of the throttle plate angle are clearly suspects in unintended acceleration. Such errors may be largely undetected in monitoring software given how late in the computation they occur. Additionally, an error in the input, particularly the accelerator pedal sensor or throttle angle sensor inputs, could masquerade as a valid sensor input and confuse the throttle controller into unintended acceleration. Table A.9-1 gives a summary of the logic models developed.

The logic model verifications identified a number of potential issues. All of these issues involved unrealistic timing delays in the multiprocessing, asynchronous software control flow. Even if they were to occur, none of these potential issues could be tied to unintended acceleration.

No cause for unintended acceleration was found from the logic model verification effort.

A.4.3 Software Algorithm Design Analysis Using Matlab Models

Model-Based Design (MBD) is a mathematical and visual method of addressing problems associated with designing complex control systems. It is used in motion control, industrial equipment, aerospace, and automotive applications.

MBD provides an efficient approach for establishing a common framework for communication throughout the design process while supporting the development cycle ("V" diagram). In Model-

Based Design, development is manifested in these steps: modeling a system, analyzing and synthesizing a controller for the system, simulating the system, and integrating all these phases by implementing the system. The model-based design paradigm is significantly different from traditional design methodology. Rather than using complex structures and extensive software code, designers can use MBD to define models with advanced functional characteristics using continuous-time and discrete-time building blocks. These built models used with simulation tools can lead to rapid prototyping, software testing, and verification. Not only is the testing and verification process enhanced, but also, in some cases, hardware-in-the-loop simulation can be used with the new design paradigm to perform testing of dynamic effects on the system more quickly and much more efficiently than with traditional design methodology.

For this study of the Toyota 2005 Camry software, model-based design techniques were applied to create high-fidelity models of the software functions and behaviors. Toyota documentation and discussions with Toyota engineering experts initiated the process. Source code analysis continued the process by increasing the accuracy of the models. And testing upon the Camry simulators and actual Camry vehicles confirmed the accuracy of the models. Efforts were made to incorporate as much actual source code into the models for further increased fidelity of the models.

No code was generated, but the models were executable, and could be run in a software environment to study function and behavior.

This MBD approach also supported the dissemination of the software functions and behaviors to the team as a whole. Presentations of the software in this manner efficiently communicated the software within the 2005 Camry microcontrollers without exposing the native source code.

Tools Used:

- MATLAB – is a product family providing a high-level programming language, an interactive technical computing environment, and functions for algorithm development, data analysis and visualization, and numeric computation.
- Simulink - is an environment for multi-domain simulation and MBD for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test systems.
- Stateflow - extends Simulink with a design environment for developing state charts and flow diagrams. Stateflow software provides the language elements required to describe complex logic in a natural, readable, and understandable form. It is tightly integrated with MATLAB and Simulink products, providing an efficient environment for designing embedded systems that contain control, supervisory, and mode logic. Models can be created of embedded software that combine logical behavior, such as fault management

and mode switching, with algorithmic behavior, such as feedback control and signal conditioning.

- SystemTest – automated model testing of Simulink models as a black box. Test values are provided to the proper model inputs; outputs of the model are tested against properties to obtain fail/pass results.
- aiT from AbsInt - statically computes tight bounds for the worst-case execution time (WCET) of tasks in real-time systems. aiT directly analyzes binary executables and takes the intrinsic processor cache and pipeline behavior into account.

A.5 Software Study

The software team performed an analysis of the Camry model year 2005 ECU (engine control unit) software to investigate if there can be plausible triggers for sudden UA in the engine control software of Toyota vehicles. As part of this study, the team also analyzed the structure and implementation of this software, using a range of different metrics. The fishbone diagram in Figure A.5-1 provides the general context for this study.

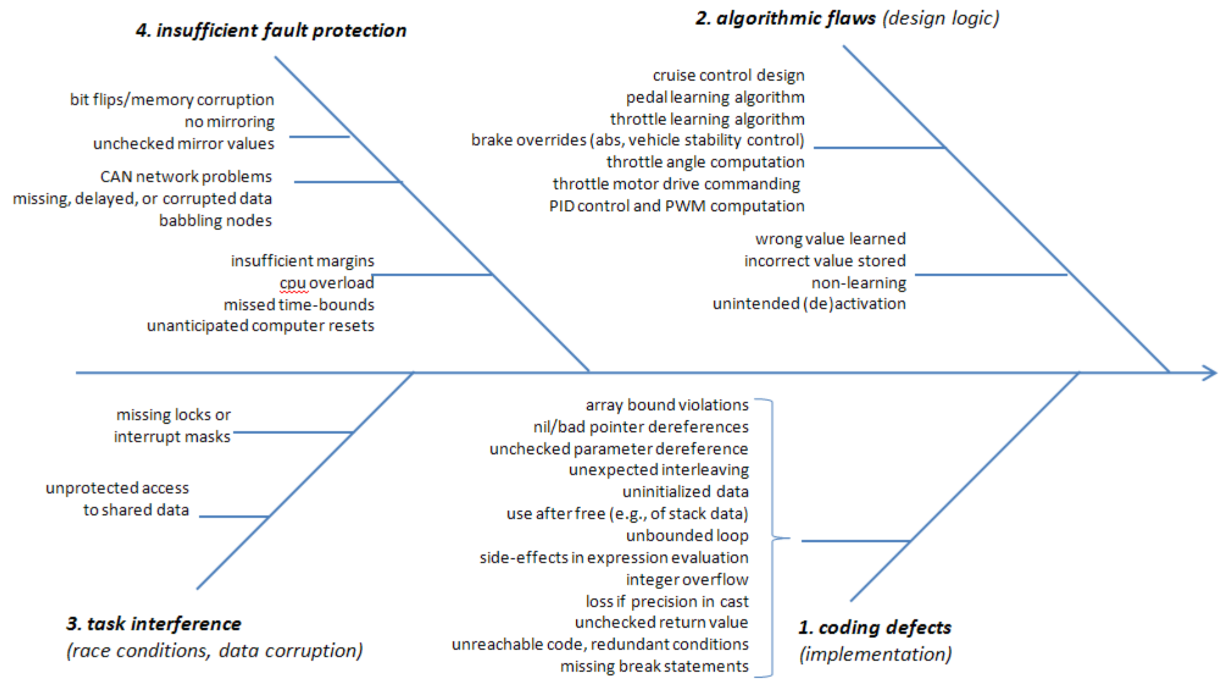

4. insufficient fault protection
bit flips/memory corruption
no mirroring
unchecked mirror values
CAN network problems
missing, delayed, or corrupted data
babbling nodes
insufficient margins
cpu overload
missed time-bounds
unanticipated computer resets
2. algorithmic flaws (design logic)
cruise control design
pedal learning algorithm
throttle learning algorithm
brake overrides (abs, vehicle stability control)
throttle angle computation
throttle motor drive commanding
PID control and PWM computation
wrong value learned
incorrect value stored
non-learning
unintended (de)activation
missing locks or interrupt masks
unprotected access to shared data
3. task interference
(race conditions, data corruption)
array bound violations
nil/bad pointer dereferences
unchecked parameter dereference
unexpected interleaving
uninitialized data
use after free (e.g., of stack data)
unbounded loop
side-effects in expression evaluation
integer overflow
loss if precision in cast
unchecked return value
unreachable code, redundant conditions
missing break statements
1. coding defects
(implementation)

Figure A.5-1. Fishbone Diagram of Potential Software Causes for UA

The fishbone diagram groups potential software causes in four broad categories. In this report, the software team addressed each of these categories:

1. Coding defects (implementation)
2. Algorithm flaws (design logic)
3. Task interference (race conditions, data corruption)
4. Insufficient fault protection

An overview of the basic operation of the control software used in the Camry05 is discussed in the following sections.

A.6 ECM Software Implementation and Basic Architecture

The ECM (engine control module) for the 2005 Camry uses a *NEC V850 E1* processor. The software for the ECU is written in ISO/ANSI C[4], and compiled for production use with the *GreenHills*[5] compiler suite version A.4.0. The code relies on the use of the *Greenhills* compiler with *pragma* directives[6] that is discussed below.

The ECM is designed to meet a range of real-time constraints for engine control. The real-time operating system used is based on the OSEK[7] standard for distributed control units in vehicles, which is supported by AUTOSAR[8] (Automotive Open System Architecture) of which Toyota is a core member. The operating system is based on the execution of tasks, each with a fixed and statically assigned priority.

The Camry 2005 code contains [redacted] Tasks that execute at fixed priority levels between 1 [redacted]

[redacted]

[4] http://en.wikipedia.org/wiki/ANSI_C
[5] http://www.ghs.com/
[6] http://gcc.gnu.org/onlinedocs/cpp/Pragmas.html
[7] http://en.wikipedia.org/wiki/OSEK, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf
[8] http://en.wikipedia.org/wiki/AUTOSAR

[redacted]

[redacted]

[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]

[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]

[redacted]

[redacted]

[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

A.6.1 The CAN Data Network

Messages sent on the CAN-bus[9] (Controller Area Network) carry identifiers of 11-bits, which means that it is compliant with the standard CAN 2.0 A base frame format. (The extended CAN 2.0 B format supports identifiers of 29 bits.) The CAN 2.0 A protocol was published as a standard by the ISO 11898 in 1993,[10] and is today broadly used in the automobile industry. The arrival of a message is signaled by an interrupt, which causes an interrupt handler to copy the data from a DMA area into a ring-buffer of 7 slots that is read and emptied by the 1msec (highest priority) task. CAN message transfers are initiated at the 250 μsec mark of the 1 msec cycle.

Signals/messages are repeated to protect against loss, but increasing the risk of network overload conditions. A node also retransmits on message transfer errors and on message collision. A babbling node, which could potentially flood the CAN network with messages, is cut-off within 65 msec at the transmitter side through a power cycle.

Occurrences of CAN data loss are recorded in SRAM, and remain available until the battery is disconnected. According to the Toyota Motor Corporation (TMC), 292 instances were reported of CAN data loss by dealers for cars brought in for any problem. [redacted]

Signals to the vehicle stability control (VSC) computer, an option on the 2005 Camry, travel over the CAN network – including on/off commands.

A.6.2 Reboot/Reset Scenarios

A watchdog control (WDC) subsystem, with separate CPU monitors the main engine control unit computer, and all the signals and events it is meant to respond to. Either the Main CPU will reset itself or the WDC subsystem will reset the ECM when abnormal conditions are detected. Some of those conditions are:

- The CPU load is too high (the idle margin time is exceeded).
- An illegal instruction is executed.
- An undefined interrupt is generated.
- An abnormal return value from a system call is detected.

[9] http://en.wikipedia.org/wiki/Controller_area_network

[10] http://www.can-cia.de/index.php?id=161

A system reboot takes about 94 ms for the Main CPU in the ECM, and 134 ms for the Sub-CPU in the WDC. This reboot time includes 30 ms for clearing RAM memory. This reboot time is fast enough that its occurrence is not noticeable when the car is in operation. The Main CPU and the sub CPU may also reboot independently, depending on the error condition that is encountered. Both the Main and the Sub-CPU generate a watchdog pulse and monitor each other's health. The occurrence of a CPU reboot is not recorded in the event-data recorders and therefore not discoverable afterwards. Independently, the power supply sub-system also monitors the WDC and can issue reset commands to both the Main CPU and the Sub-CPU in the WDC.

A.7 Software Implementation Study

The software development process followed by Denso is based on a five step process.

- Specification
- Design
- Implementation
- Unit test
- Integration test

Note that there is no intermediate phase reserved for sub-system testing, e.g., of the cruise control sub-system completes unit test, and moves on to integration test, without an intermediate sub-system test.

The system design and software specification is performed by TMC engineers. The detailed software design, implementation, and unit-tests are done by DENSO engineers. After delivery of the code to TMC, acceptance tests (the integration test) are performed on a special high-fidelity hardware testbed called *MITY*.

A strict convention is used for naming variables, tables, functions, and files. The suffix *_l_* indentifies a logic component, *_c_* identifies constant data (e.g., as recorded in tables), and *_x_* execution. A filename prefix is used to identify *e_* engine control, *w_* diagnostic code, *g_* application platform code, *ge_* engine control unit layer code, and *gc_* cpu layer code.

Every variable name carries a 3-letter prefix that identifies its scope and type. For instance, the prefix *s2t_* identifies a signed variable (*s*) of two-bytes (*2*) with local scope (*t*). Similarly u4g_ identified an unsigned variable of four-bytes with global scope, and s2s_ stands for a signed two-byte variable that is declared file-local (static). Variables can also have suffixes such as *_map* or *_tbl*.

In addition to the *MITY* testbed mentioned above, Toyota reported the use of a number of tools and checks to assess the quality of their software.

- *QAC*, a tool from the British company Programming Research[11], is the primary tool used to catch common coding defects and to verify compliance with some commonly used coding rules.
- *CAST* is an in-house TMC tool to verify type conversions and potential cases of value overflow and some common cases of coding defects. The tool checks compliance with 25 specific rules (e.g., possible incorrect use of precedence rules in expressions with binary shift operations, or the potential loss of precision in an expression with mixed variable types).
- *Careless*, another in-house TMC tool for verifying different coding patterns and coding style, perhaps compliance with naming conventions (not much else is know about this tool and the team was not able to obtain a manual with more information).
- A *Mode Transition* checker, to verify correct transitions between normal and failure modes.
- A *Stack overflow* checker. The system stack is limited to just 4096 bytes, it is therefore important to secure that no execution can exceed the stack limit. This type of check is normally simple to perform in the absence of recursive procedures, which is standard in safety critical embedded software.

  A *Task interference* check, which is a mostly manual check, assisted by a tool-generated list of cases to be inspected. This is covered in greater detail in the Section A.8.2 Access to Shared Global Variables.

Some basic statics on the source code are summarized in Table A.7-1.

SLOC – Source Lines of Code are compiled into executable code.
NCSL – Non-Commented Source Lines have no additional explanatory text.
Source File – Source code file complied into executable code.
Header File – Source code file used to describe source code interfaces.
Comments – Explanatory text in addition to SLOC.

A comment ratio (Comments/NCSL) near or above 1 can indicate unusually densely commented code. It should be noted, the source lines were readable as C, and the comments were in Japanese.

[11] http://www.programmingresearch.com/qac_main html

Table A.7-1. Basic Code Size Metrics Camry05 Software

C code	#Files	SLOC	NCSL	Comments	NCSL/ File	SLOC/ NCSL	Comments/N CSL
sources	1,761	463,473	256,647	241,683	145.7	1.8	0.9
headers	1,067	100,423	39,564	67,064	37.1	2.5	1.6

As part of this study, the team also analyzed the structure and implementation of this software, using a range of different metrics.

The Toyota code was developed and implemented using coding rules and standards matching the Toyota coding rules and standards.

A.7.1 TMC Coding Rules

A specific set of coding rules was defined for the TMC/Denso ECU software development. The rules are described in a document titled: *Power train system electronic control unit control software for 32 bits -- Coding rule*.[12] The full document has 81 pages and has ten main sections, as shown in Table A.7-2.

The larger part of the document concerns naming conventions and coding style. Most relevant for this study were the specific coding rules. Many of the rules that are articulated in the document are occasionally violated in the code. Examples of rules that were generally complied with include:

- Do not use tab characters, limit lines to 80 characters, and indent lines in 2-space increments.
- Do not use octal numbers.
- Do not use the multiplication, division, or modulo operator, instead use library functions which are coded in assembly language for speed.
- Do not use function calls in *if* conditionals (to avoid possible side-effects).
- Split compound expressions across lines.
- Place the opening curly brace of a block on same line as an *if*, *while*, or *for* statement.
- Check bounds with multiple if-statements in series not with if/then/else sequences.
- Use only one return statement per function.

[12] As translated from Japanese to English by the *Atlas* tool that Toyota provided to us; http://www.fujitsu.com/global/services/software/translation/atlas/

Table A.7-2. Structure of TMC Coding Rules Document

Section	Title
1	Document version history
2	Table of contents
3	Introduction
4	Data types
5	Functions
6	Header files
7	Include files
8	Commenting conventions
9	File structure
10	Coding Rules (p.53-81)

The following Toyota rules in the Toyota document are frequently violated in the ECM code. A violation of these rules does not automatically imply risk, but it can affect code clarity, consistency, and maintainability.

1. Use parentheses in expressions for disambiguation.
2. Consistently use symbolic names for variables and constants.
3. Use the *sizeof* function instead of a constant.
4. Use parentheses around the body of a macro definition.
5. Terminate each *switch* case with a *break* statement. (Many violations were noted, without comments indicating that the fall-thru to the next case statement is intended)
6. Do not use ternary operations (i.e., statements of the form *x?y:z*) in the *etcs* module. (count was **211** uses of ternary operations in the code, but only one appears in the *etcs* module.)
7. Do not use pre- or post-increment or decrement operators in assignments (e.g., *x= y++*). (Counted **61** violations of this rule in the code.)
8. Limit variable names to 31 characters. (Counted **58** violations in the code. The longest names have **36** characters, e.g.: *u2s_vpdccstdrnlrn_gnslpavcstdrnl_tbl*. Possibly, the *u2s_* type prefix and the *_tbl* suffix were added late in development.)

9. Switch statements must include a default. (Counted **343** *switch* keywords in the code, and **238** uses of the *default* keyword, which means at least **105** violations of this rule.)
10. Terminate comments on the same line where they are started. (Counted **206** violations in the code, including typos like /* ... /* in multi-line comments.)
11. Include directive must specify pathnames for include files starting with one of only three allowed prefixes: "./", "../", or "<". (Counted **37** violations of this rule, **15** of which appear in the file *os/kernel.h* which was likely excluded from the TMC coding rules.)

The rule about references to include files (the last entry in the list) is of special interest because the use of relative pathnames introduces a risk of accidentally referring to an incorrect file when a source file is moved within the file system hierarchy during development.

Not phrased as an explicit rule, but complied within the code, is the absence of dynamic memory allocation. In the TMS software, all memory and all data objects are statically allocated at system boot time. This practice reduces issues with the allocation/deallocation of memory and references to dynamic memory during execution.

A.7.2 NASA Mission and Safety Critical Software Coding Rules

There are a number of rules that are commonly applied to NASA mission and safety critical software but were not required in the TMC coding rules.

- *Union* data structures. The use of *union* data structures is generally prohibited in safety critical code (e.g., in the MISRA[13] coding guidelines), but the TMC coding rules document explicitly allows its use.

  156 separate uses of *union* data structures were counted in the Camry05 code, compared with **229** uses of the, generally safer, *struct* declarations.

- *Excessively long functions*. A common practice is to allow no functions longer than 60-75 lines to enhance manual code inspections.

  200 functions longer than 75 lines in the code were counted, with the longest functions spanning several hundreds of lines.

- Use of the *#undef* compiler directive, to override earlier macro definitions. This allows definition to change within the source code.

[13] http://www.misra.org.uk/

2,659 uses of *#undef* in the Camry05 software were counted.

- Compiler *pragmas*. A compiler *pragma* can be used to leverage a non-language standard feature supported by a specific compiler, and thereby makes the code implementation dependent (or more precisely: compiler dependent). *Pragmas* can introduce errors when a different complier is used to compile the source code.

  2,339 uses of *pragmas* in C source file and **472** uses in C header files were counted. These uses are of **14** different types and are mostly used to link data structure names to specific areas of DMA memory.

- *Guaranteed loop bounds*. Another rule absent from the coding rules document but generally present in coding standards for safety critical software is a precaution against unbounded loops.

The software team performed additional, more detailed analysis of the code with the help of a range of static source code analyzers, which is discussed in the next section.

A.8 Static Analysis Results

This section summarizes the results of a static analysis of the Camry05 code with the following tools.

- The standard *gcc*[14] compiler version 4, both in basic mode and in strict language compliance mode with all warnings enabled.
- *Coverity*[15] version 4.2, with a range of additional Extend checkers to verify compliance with commonly used coding rules to reduce risk.
- *Codesonar* [16]version 3.6p1, a version that includes support for JPL Coding Rules used for mission critical flight software development.
- *Uno*[17] version 2.12, for additional checks.

[14] http://gcc.gnu.org/

[15] http://www.coverity.com/

[16] http://www.grammatech.com/products/codesonar/overview.html

[17] http://spinroot.com/uno/

A method for identifying potential problems involves analyzing compiler warnings when run in either basic or pedantic mode.

Compiler Analysis

The metric used here is the number of warnings issued per one thousand lines of code, after stripping comments and blank lines from the code (abbreviated as *KNCSL*). The few warnings issued for the Camry05 code can be attributed to minor differences in the parsers of the different compilers used: some compilers can be less forgiving for small ambiguities in the language definition (e.g., the placement of braces or the separation of tokens). The 142 warnings issued by *gcc* in strict mode, by warning category, are summarized in Table A.8-1.

Table A.8-1. Gcc-strict Warning Categories

Number of warnings	Category
52	Unused variable
33	Use of /* within a comment
21	Declared static but never defined
19	Defined but not used
9	Value computed not used
2	Missing braces around initialize
2	Function pointer cast to object pointer-type
1	May be used uninitialized
1	Suggest parentheses around && within ||
1	Overflow in implicit constant conversion
1	Static declaration follows non-static

The warning overflow of implicit constant conversion was generated for the assignment of the hexadecimal value 0xff (255 in decimal) to a variable of type signed character, which has a value range 0..128 and -1..-127. The suggested use of parentheses in an expression containing both logical and (&&) and logical or (||) operators is for a preprocessor directive of the following type "*#if X && Y && Z* || *A*" which indeed requires careful familiarity with the precedence rules used by the C preprocessor to determine if its intent matches the actual interpretation.

Very few compiler warnings were generated by the MY2005 Camry source code. No issues leading to a UA in the TMC system were found when these compiler warnings were reviewed and inspected in the source code.

Static Analysis

The next series of measurements was done with two state-of-the-art static source code analyzers. Consistent with the team's experience, there is no significant overlap between the warnings issued by the two tools, which is a strong motivation for the use of more than one strong static source code analysis tool in software development. At the current state-of-the-art, the leading tools all have different strengths and should be combined for a thorough analysis. The software team intentionally applied analysis tools not used earlier on this code. (Internally TMC relies on the use of the *QAC* [18] tool, and the firm Exponent hired by TMC to perform additional analyses used the tool *Polyspace*,[19] two other leading tools in this domain.) As can be seen in the reports, the Coverity tool tends to be more conservative in the generation of reports, while the Codesonar tool does a deeper (and generally slower) analysis, and generates more reports.

The company Coverity regularly performs a scan of public-domain software and makes the resulting metrics available to developers. For the *gcc* compiler (with 851,149 lines of code), the published metric is currently **0.219** Coverity warning per *KLOC*[20]. The measure *KLOC* –short for *one thousand lines of code* – may differ from the team's metric *KNCLS* in that it is likely to be a raw line-count that includes comments and blank lines.

The MY2005 Camry Coverity analysis indicated 1.41 warnings per thousand lines of code. If measured per raw lines of code, the count for MY2005 Camry would become **0.74**. Any count near or below *one warning per one thousand lines of code* generally indicates a well-managed software development process.

The **418** warnings issued by Coverity, grouped by warning category, are summarized in Table A.8-2. A similar listing for the 2272 warnings issued by the Codesonar tool is summarized in Table A.8-3.

[18] http://www.programmingresearch.com/qac_main html

[19] http://www.mathworks.com/products/polyspace/

[20] http://scan.coverity.com/rungAll.html

Table A.8-2. Coverity Warning Categories

Number of warnings	Category
209	checked return
97	declared but not referenced
39	Overrun static
19	Integer overflow
17	Potentially unbounded loop
13	Uninitialized variable
8	Dead code
6	No effect
5	Missing break statement
5	include recursion

Table A.8-3.Codesonar Warning Categories

Number of warnings	Category
2272	Global variable declared with different types
962	Buffer overrun
333	Cast alters value
142	Redundant condition
99	Condition contains side-effect
92	Code before #include
64	Multiple declaration of a global
63	Unreachable computation
63	Useless assignment
22	Uninitialized variable
21	Unreachable call

Number of warnings	Category
16	Unused value
14	Buffer underrun
14	Empty if-statement
12	Multiple statements on one line
12	Unreachable data flow
11	Type underrun
6	Unreachable control flow
2	Macro defined/undefined in function body

No issues leading to a UA in the TMC system were found when these static code analysis warnings were reviewed and inspected in the source code.

A.8.1 Additional NASA/JPL and MISRA Coding Rules

The next metric the software team collected is related to a series of coding rules developed and used at NASA/JPL for flight software. These rules are inspired by in-flight anomalies and risk-related defects in code. In this context the software team also looked at the compliance with a selection of the MISRA coding guidelines, which are recommended for use in automotive software. In discussions with TMC the software team learned that the MY2005 Camry coding rules used by Toyota predate the MISRA guidelines (the original MISRA coding guidelines date from 1998), but that an estimated 50% of the MISRA rules were being followed.

The coding rules used by TMC and DENSO are described in more detail in the following section of this report. For the next measurements the software team looked at a subset of the rules that could be verified with an extended version of a preprocessing tool developed at NASA/JPL for this purpose. (This means that most of the rules checked with this tool are related to the conforming use of preprocessing directives and macro definitions.)

The Toyota code was developed and implemented using coding rules and standards matching the Toyota coding rules and standards. As a result, the static analysis tools used in this study, configured for NASA/JPL use, produced more warning reports when used on the Toyota system.

The deviations reported in the MISRA rule check are categorized as follows. For each case, the software team provide the number of reports issued in the first column, followed by the MISRA rule number and a brief description of the rule itself. A total of 14 of the 35 rules the software team checked had reported deviations in the Camry05 code.

Table A.8-4. Deviations from 35 MISRA coding rules

Number of warnings	Rule number	Description
2763	19.7	Use of a function-like macro instead of a C function
2659	19.6	Use of the directive *#undef*
1615	19.4	Possibly non-compliant macro format
29	19.11	Evaluation of an undefined macro name
19	2.3	Nested /* comment
14	19.12	Use of more than one token-pasting operation (##) in a macro
13	19.16	Use of a macro call with insufficient parameters
7	18.4	Use of a *union* data structure
4	14.5	Use of the keyword *continue*
4	19.5	Use of a macro definition within block scope
3	19.10	Formal macro parameter not enclosed in round braces
2	19.4	Use of a semi-colon at the end of a macro body
1	19.13	Use of token pasting operations (##) in a macro definition
1	19.1	Include directive (*#include*) is preceded by code

The software team also checked compliance with a small set of coding rules that was identified in NASA/JPL flight software development as strongly related to code safety. The warnings issued for compliance with these so-called "*Power of 10*" Rules[21] can be grouped into the categories summarized in Table A.8-5.

[21] "The Power of Ten -- Rules for Developing Safety Critical Code," *IEEE Computer*, June 2006, pp. 93-95. http://spinroot.com/p10/

Table A.8-5. Deviations from Power of 10 Rules

Number of warnings	Description
6,971	Scope could be local static
1,086	Scope could be file static
502	Unchecked parameter dereference
425	Parameter not checked before use as an index
326	Parameter not checked before dereferencing
200	Functions longer than 75 lines NCSL
59	Potentially unbounded loop
18	Pointer type inside typedef
16	Potentially recursive macro

The first two entries, with the largest number of reports, correspond to variable declarations that have larger scope than is necessary.

The length of a function is generally limited to 60-75 lines of code, after stripping blank lines and comments. There are ***200*** functions in the Camry05 code that exceed this length. The longest function far exceeds this limit, at ***740*** lines NCSL.

The reports for 16 potentially recursive macros flags the following type of macro use in the code.

```
#define x(y)          x[y]
#define x(y)          x_tmp(y)
 #define x_tmp(y)     x##y()
```

In the first example, a macro is used to provide an alternative syntax for an array index; allowing the programmer to write *x(y)* instead of *x[y]*. The use is less than optimal since it makes the code look as if a function call, or equivalent, is used, where in fact an array index operation is performed.

In the second example a temporary macro name is used to provide a token-pasting operation that allows the programmer to write *x(y)* when in fact the code will execute *xy().* The use of these patterns affect code clarity and readability.

Potentially unbounded loops are loops that have no maximum loop count or limit that guaranties the loop terminates.

The warnings issued by the *Uno* tool are categorized in Table A.8-6.

Table A.8-6. Uno Warning Categories

Number of warnings	Description
453	Missing else
89	Possibly uninitialized variable
7	Array indexing error (in unexecutable part of conditional expressions)
2	Array of 16 bytes initialized with 17 bytes

The majority of the deviations detected by *Uno* (***453*** out of 551 reports) are cases where *if-then-else-if* series of statements are not terminated by a final *else* clause. This rule is similar to the common coding rule (also included in the MISRA guidelines and in the TMC coding rules) that every *switch* statement must contain a *default* clause. Another Uno report flagged the initialization of an array of *16* bytes with *17* bytes of data, causing a one-byte overflow. The software team confirmed in the code the overshoot had no harmful effects on execution. For the risk-related rule set, low numbers are understandably preferred.

In summation, the Toyota software development process uses an internal coding standard that does not contain all the comparable rules and guidelines contained in NASA/JPL and MISRA coding guidelines.

A.8.2 Access to Shared Global Variables

The Camry05 software code structure relies on the use of a single large memory space that is shared among all tasks, with unrestricted access. This is in contrast to system designs where each task is given strictly private memory that is not accessible to other tasks, and very limited amounts of memory are used for shared data[22]. Strict access patterns (e.g., using interrupt masks, semaphores, or locks) are then used to secure the safety of access to shared data. In this code structure, the scope of all data object (the part of the code where the object is visible and accessible) is normally restricted to a minimum. The underlying principles used in such systems are based on firewalls and containment, limiting cross-coupling and inter-task dependencies to a

[22] Following, for instance, the ARINC 653 standard http://en.wikipedia.org/wiki/ARINC_653 used in the airline industry.

minimum. Reduction in variable scope allows implementation errors to be detected when the code is compiled, and when the code executes.

In the Camry05 software these principles are not used, which requires more manual inspection and allows for less automated analysis of the correct protection and access to these variables during development. To get an impression of the scope of the task, the software team counted the number of externally visible global variables, as reported by the standard Unix tool *nm*, which examines the executable image of code. The nm tool reports the names of objects in eight different categories, e.g., depending on whether an object is initialized or uninitialized, is externally visible or file-local, was declared constant, or defines text. From its output it can also be determined how many names for different objects (i.e., located at distinct addresses in memory) carry identical names (i.e., the use of name overloading for distinct objects declared in different scopes). The results of this analysis can be summarized as follows.

31 names are defined multiple times in different scopes. The most frequently reused name is *sts_flags1*, which appears in 57 different scopes in uninitialized static (i.e., file-local) declarations. The majority of names is declared in just a single scope, as summarized in Table A.8-7.

Name aliasing is discussed in more detail below.

The data objects can be grouped into standard categories, as shown in Table A.8-8.

Table A.8-7. Variable Name Reuse

Camry05	Flight Software Reference	Description
31	55	variable names defined multiple times in different scopes
57	8	largest number of uses of a single variable name
16,354	56,482	variable names declared in just one single scope and not reused

Table A.8-8. Variable Scope

Camry05	Type	Description
1,872	b	Uninitialized static (file-local)
2,800	C	Uninitialized common (extern)
108	d	Initialized static (file-local)
6,473	D	Initialized common (extern)
5	r	Read-only static (file-local)
91	R	Read-only common (extern)
914	t	Text, static (file-local)
3,710	T	Text, common (extern)

In the Camry05 software a majority of all data objects (***82%***) is declared with unlimited scope, and accessible to all executing tasks. The relevant usage is illustrated in Table A.8-9.

To determine the correct access to each of the shared data objects in the Camry05 code, the complete system of all tasks would need to be inspected~~.~~

Table A.8-9. Use of Global Scope

Camry05	Type	Description
9,273	C+D	Externally visible variables
1,980	b+d	File-local variables

The coding pattern recommended in the coding rules document is to store the value of a shared global variable into a (non-shared) local variable before using it in computations, and writing a new value back to its shared location just once, at the end of the computation. This rule is to be followed unless the global variable is accessed just once in the computation.

Coding rule 651 of the TMC document describes in more detail what the recommended use of interrupt masks is. The rule states that interrupt masking should not be used when two tasks run at the same priority level, only to protect a lower priority task from interference by higher priority tasks.

This rule is based on the fact that equal priority tasks cannot interrupt each other. Both can still be interrupted by a higher priority task. If because of this interruption the second task does not complete, and the first task restarts in the next time interval, it could still overwrite the result of the interrupted second task.

If two tasks running at different priority levels access the same data, then the lower priority task must use interrupt masking to protect against interference from the higher-priority task.

This rule is not always followed in the code. In a few cases, the lower priority task merely sets a flag before entering its critical section and the higher priority task checks this flag before accessing the same data.

There are cases in the code where tasks of different priority levels (e.g., levels 14, 12, and 4) access the same global variables without using interrupt masks (pattern: read, store local copy, update, write new value). An example is the use of variable *s2s_eafsfb_gaind*, which is declared as a *non*-volatile static variable. This use would appear to be in violation of coding rule 651. All *constant* global variables though are consistently declared *volatile*, in compliance with the coding standard. Although a specific reason for the use of *volatile* for constant data is not given, it could be that the use of the volatile qualifier makes it possible to patch values in a running system (because the compiler can then not optimize away the variable references and inline the constant values).

Despite the rigor in the use of the *volatile* qualifier on *constant* data, other shared global variables are not always declared *volatile*. The team counted **11,528** non-constant, shared global variables in the code.

There are only **865** uses of interrupt masking in the code, in **194** different source files. This indicates that access to global variables is not always done under protection of interrupt masks.

There are many cases of nested interrupt masking. For example, *gadsl_out()* masks interrupts and calls *gedsl_drive()* which also masks and then re-enables interrupts. The team modeled the specific method for interrupt masking method that is used in a Spin model (*di.pml*), and verified that the protection with nested calls is correct.

[redacted]

[redacted]

[redacted].

[redacted]

[redacted]

[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

[redacted]

[redacted]

[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

[redacted]

The software team adapted one of their tools (the static analyzer *uno*) to track which tasks can access each shared global variable, specifically which tasks used (read), update (write), or take the address of a shared variable. The resulting list of accesses was grouped by the *number* of different tasks that access each unique variable, as shown in Table A.8-12, Shared Global Data.

Table A.8-12. Shared Global Data

Number of shared variables	Number of different Tasks accessing each variable
909	2
365	3
104	4
67	5
3	6
6	7
1	8
5	9
2	10
4	12
7	13
6	14

The largest number of tasks accessing the same shared global variables consists of 14 tasks. The task *T6*, which executes at a relatively high priority level of 16, accesses a large number of global variables, and appears in 8 of the 12 groups of tasks in Table A.8-12.

The six variables accessed by the largest number of tasks are:

Access	*Variable Name*
W	*u2g_gasram_read_u2 u1s_gasram_mirror*
R	*u2g_gasram_read_u2 u1s_gasram_mirror*
R	*u2g_gasram_read_u2 ptg_gasram_u2db_tbl*
R	*u2g_gasram_read_u2 ptg_gasram_u2init_tbl*
R	*u2g_gasram_read_u2 u2g_gasram_u2dbmax_tbl*
R	*u2g_gasram_read_u2 u2s_gasram_keyword*

The variable *u2g_gasram_read_u2 u1s_gasram_mirror* is the only variable in this group to appear both in update and in read access scenarios (by all 14 tasks). All other variables are read, but not updated, by these tasks. The group of tasks that share access to these six variables, and the priority levels at which they execute are:

Task	Priority
T1_1msec	20
T5	18
T6	16
T7	16
T9	16
T11	16
T13	16
T15	14
T17	12
T19_4msec	10
T20	7
T23	4
T22_8msec	4
T24_idle	1

In a full analysis each variable access must be inspected and verified to be safe, taking into account possible use of interrupt masking, and a comparison against the rules stated in the coding standard document. Toyota confirmed to the team that this was done for the Camry05 code with a manual inspection process, with the results documented in an annotated list of verified access patterns. By the team's estimate a thorough manual verification process could consume up to 17-34 weeks (5-10 minutes per variable, with 9,273 shared global variables to consider). A significant manual inspection effort would be required to assure correct access to these shared global variables.

In this study, inspections were performed on a small number of randomly selected tests and found no discrepancies. There are several remedies that would be available to reduce the effort on verification of global variable access:

- Reducing the amount of global data (the preferred method and consistent with improved code structure).
- The development of a specialized tool that can reliably mechanize the analysis.

There are many cases in the code of near duplicate copies of functions or parts of functions. The software team also noted significant inconsistency in how, what are essentially the same, functions are coded in different parts of the code. In some cases entire files are near copies of each other (for instance the files *gapa1gx_mat.c* and *gapa2gx_mat.c*). Code duplication is a maintenance problem when changes must be applied and tracked across multiple copies of code.

Small deviations were also noted in near-duplicate files. (For instance, as a single difference other than a replacement of all 1-suffixes with 2 suffixes, one file differs from its near-duplicate only in the use of variable name *s4t_gavpaad* instead of *s4t_gapa2trn*). The near-duplicate files are best generated from a single source, or the code could be merged and generalized.

Ubiquitous use of global variables gives significant problems in limiting scope and performing analysis of sub-systems. In the pedal learning subsystem for instance (which consists of nine C source files) no less than 1,255 external global variables are referenced, including 130 tables of pointers to various types of global data. The unit-test for the main file has 35 arrays with sequencing data (indicating many potential dependencies with other parts of the code).

A.8.4 Name overloading/aliasing

The TMC coding standard defines a specific naming convention for variables. However, many variables are renamed and referenced by a different name (aliasing) throughout the code. Some macro names are defined in multiple (and different) ways in different source files (e.g., the name *NOT_EXIST* is sometimes defined to be 0 and sometimes 0xff). A reference to a single global variable can appear under many different names – sometimes a name is redefined via macros, sometimes it is redefined via several layers of indirection through pointers in tables.

The frequent overloading and redefinition can make it difficult to trace which data object is being accessed. Example:

> The file *src/pf/ecu/gesgm2drv_mng_mat.c* contains calls to macros named *enter_xxx_task(ON)* and *enter_xxx_task(OFF)* with *xxx* equal to *rcva*, *rcvb*, *com*, *dma-int*, and *4msm*. The macros are used to set and clear five different bit-fields, named *sts_flags.b0* through *sts_flags.b4*. The same fields are also accessed under different

names via macro definitions in four different files that introduce the following name mappings (suggesting that different objects are accessed, but in fact all accessing the same shared object):

enter_4msm	::	bis_4msm
bis_4msm	::	(sts_flags.b0)
sts_flags.b0	::	bis_af_xinka
sts_flags.b0	::	bis_explsout_rev

The last two names are also used independently of the *enter_xxx_* macro to set or clear the bit-fields, making it extra difficult to track the access patterns.

As a final check, the code defines *four* different copies of a data object named *sts_flags* (in files *gedsl_mat.c* and in *gedclx_mat.c*). This means that the single name *sts_flags*, which can be accessed under several different aliases, does not always refer to the same object.

The software team considered the potential for name confusion that can be caused by macro redefinition of the same name. In this evaluation **2,352** cases were counted where the right-hand side of a macro occurs in at least one other macro definition unchanged. Specifically, there are:

659 cases where there are at least 4 different redefinitions of the same macro

300 cases in these 659 cases which are non-trivial, (i.e., non constants)

Among the last **300** cases the following examples are highlighted.

1,002 uses of the name *u2s_WMEMCNT_NO_EEPROM* in *different* macro definitions as the (complete) right-hand side. This means that this one object or quantity is known under 1,002 different aliases.

148 uses of the name *u1g_vlsb_1degc_p(y)* appears in *different* macro definitions as the right-hand side.

101 uses of the name *GCMAP_NONE* appears in *different* macro definitions as the right-hand side.

A frequently used name is also the structure name *sts_flags1*. Different fields in this structure make many distinct appearances as the *right-hand side* (the defining body) of macros:

46 definitions of *sts_flags1.b0*.

36 definitions of *sts_flags1.b1*.

32	definitions of *sts_flags1.b2*.
25	definitions of *sts_flags1.b3*.
13	definitions of *sts_flags1.b4*.
10	definitions of *sts_flags1.b5*.
6	definitions of *sts_flags1.b6*.
3	definitions of *sts_flags1.b7*.

The software team noted earlier that the name *sts_flags1* itself also appears at **57** distinct addresses in the symbol table, referring to equally many distinct data objects.

The frequent use of code duplication, name overloading, name aliasing, and redefinition can impact code clarity, maintenance, and analysis.

The Toyota software makes extensive use of name overloading, indirection, and code duplication. This affects code structure and code clarity, and can make problems hard to diagnose and fix. It can also hamper software analyses.

A.8.5 Dead Code

Dead code, i.e., code that is unexecutable in a particular version of the code (e.g., model year 2005) can also make code maintenance and analysis more difficult than necessary.

An example is the following fragment of code.

File *wmemcnt_mat.c*:1009:

```
s4t_rslt = (s4) ERROR;          # ERROR == -1
if (s4t_rslt == (s4) OK)        # OK == 0
{       if (u1t_result == ...)
```

In this fragment the value of a signed four-byte variable is first set to negative one. In the immediately following line the same variable is tested for equality to zero. Because the value cannot be zero at this point, the subsequence part of the code becomes unexecutable (dead code). The fragment of code contains a reference to a variable that in this version of the code remains uninitialized at this point, which triggers two separate static analyzer warnings: one for the inclusion of dead code and the second for the potential use of an uninitialized variable. Dead code is normally not allowed in the MISRA guidelines.

Note, the unused code is compiled and exists in memory at execution. The signed four-byte variable also exists in memory and could be altered during execution, in essence, enabling the code to execute.

A.8.6 Data Mirroring of Persistent Parameters

A small set of critical parameters is saved in two separate places in SRAM storage, though possibly contiguously in the same memory bank or page. The two copies are mirrored to increase the level of protection that is obtained in this way. This means that all the bits in one of the two copies stored are inverted from the other copy. [redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

A.9 Software Logic Model Checking

Logic model checkers use efficient algorithms to explore all possible executions of a system in an attempt to locate those executions that violate user-defined logic properties. The exploration is in principle exhaustive and thus if it completes without finding incorrect behavior then the system is proven correct. Because the exploration is exhaustive, model checkers can excel in finding incorrect behavior in 'corner cases' which are exactly the types of errors that tend to occur infrequently and be overlooked in even rigorous standard software testing. Key to a logic model is the formulation of the model's behavior and its correctness.

The ideal logic model for UA would be a model of the complete throttle control system. Such a model would have 'throttle plate angle' as a single output, a variety of sensors values as input, and a core correctness claim relating the throttle plate output to the sensor inputs. Notable inputs would be the sensed accelerator pedal position and the sensed throttle plate angle. A specific correctness claim would be a kin to 'the throttle angle should never be wide open unless the accelerator pedal is fully depressed.' A violation of such a claim would be a strong candidate for UA.

To construct a comprehensive model requires a faithful representation of an environment model that can capture and reproduce all relevant inputs to the software being analyzed. For a modern car, this can include a large array of input variables that can all influence the behavior of the vehicle, beyond the obviously relevant inputs of the accelerator and brake pedal positions.

Often this type of environment model is available from existing simulation tools that manufacturers of embedded systems commonly use to analyze complex software systems before deployment. No such simulation environment, though, was available for this study. Toyota colleagues stated that no simulation tests of this type were performed in the development and testing of the Camry05 software. The scope, size and complexity of the Toyota throttle control software and the limited duration of the study prevented the team from attempting to create a faithful environment model (such an effort can take a significant amount of time, if the information has to be constructed from scratch).

The logic models developed focused on those modules within the Toyota throttle control software that were deemed, based on the team's study, most likely to contribute to UA. The modules selected are related to:

- the conversion of the desired throttle position into a pulse width modulated set of H-Bridge transistors signals;
- the generic processing of the sensors inputs of any type; and
- the specific processing of the accelerator pedal position sensor input.

This set of logic models are involved with the input/output of the throttle control software. Possible errors in modules related to the output of the throttle plate angle are clearly suspects in UA. Such errors may be largely undetected in monitoring software given how late in the computation they occur. Additionally, an error in the input, particularly the accelerator pedal sensor or throttle angle sensor inputs, could masquerade as a valid sensor input and confuse the throttle controller into UA.

In what follows, each of the logic models is described. The description includes the purpose of the logic models, the correctness claims applicable to the model, the result of the model verification and the conclusions drawn from the model and its verification.

Logic model checking follows the following steps. A model of the software system, possibly including the actual source code of the system, is developed for model checking. Specific claims are developed concerning the property or attribute to be checked. The model and claim are input to the model checker. The model checker determines every path through the software system, and checks the claim on every path. The model checker reports the paths that exist where the claim is not met.

The unique aspect of this form of model checking is the automated check of every path through the software system. This form of analysis requires processing speed and increased memory space to perform the check on every path through the model of the software.

The outcome for each model checking run can be:

Verified: The claim made upon the model being checked is not violated on any path checked in the model.

Not verified: The claim made upon the model being checked is violated on at least one path checked in the model.

Inconclusive: The model and claim could not be completely analyzed.

Note that any model checking claims that indicated a possible influence upon a UA influenced the design of the tests performed on vehicle hardware. Table A.9-1 gives a summary of the logic models developed.

Table A.9-1. Summary of logic models

Model	Type	Conclusion
Interrupt Enable/Disable Pairing	Computation	Verified
Accelerator Pedal Learning	Computation	Inconclusive
Sensor Input	I/O	Potential Issue
Motor Drive IC	Computation	Verified
Port Register Input	I/O	Verified
PWM Functionality	Computation	Potential Issue

A.9.1 Interrupt Masking Method

Purpose: Verify that CPU interrupt enabling and disabling method used in the Camry05 code correctly protects access to shared global data, also when interrupt masking is used recursively.

System: A region of software code that must execute atomically is known as a *critical region* or *section*. Mechanisms to protect different tasks from interfering with each other's computations, and possibly corrupting data, conventionally include the use of *semaphores* and mutual exclusion *locks*. On a single processor CPU, a mutual exclusion lock can also be achieved by temporarily masking the interrupts. By masking a clock interrupt, it can be assured that the currently executing task cannot be preempted by higher priority tasks. Clearly, these interrupt masking intervals should be as short as possible, to avoid jeopardizing real-time performance and responsiveness to external events. The method can carry low overhead and is therefore sometimes preferred in embedded systems code, but it carries a risk that the protection does not span sufficiently many operations.

In the Toyota code it was noticed the use of recursively nested interrupt masking, which places additional requirements on its Implementation. It should, for instance, be impossible for a nested interrupt masking region to re-enable interrupts that the higher-level task, upon return from the nested call, assumes still to be in effect. Not all implementations provide guarantees for nested operation of interrupt masks.

Model Description: The software team developed logic model based on the implementation of the *glint_ei()* and *glint_di()* operations that are used in the Camry05 code for interrupt masking, shown in Figure A.9-1.

The model contains seven macro definitions to capture the relevant part of the interrupt masking functionality used. The key operations are *DI()* (Disable Interrupts) and *EI()* (Enable Interrupts), which are invoked by the functions *glint_di()* and *glint_ei()*. The current state of the processor status word (PSW) is stored in a local variable, which is modeled here with the help of a macro token-pasting operation.

The execution of two tasks of arbitrary priority levels is modeled with the help of a *proctype* specification *task*, instantiated twice with the help of the *active* prefix. Any other number than two can also be used, without a change in the outcome of this verification. The tasks are designed to recursively invoke the interrupt masking routine and count how many executing processes can simultaneously reach the inner-most point critical region in the code. A logical assertion then verifies that this number can never be greater than or smaller than one. At the end of the execution, when the interrupt mask is removed by both processes in both nested regions, a second assertion verifies that the count is zero and interrupts are correctly re-enabled.

Claims w/ Results: The model checker verifies the correctness of this model in a fraction of a second, conclusively proving that the implementation is correct under all possible behaviors of the scheduler, and independent of task priorities.

Result: Verified. The claim made upon the model being checked is not violated on any path checked in the model.

The Toyota use of *glint_ei()* and *glint_di()* is sound for use in protecting critical regions, including the use in nested interrupt masking regions. This model rules out critical region race conditions as a possible source leading to UA, but only for those instances where interrupt masking is used.

Result: The method used for masking of recursively enabled and disabled interrupts cannot lead to race conditions in critical sections.

```
#define u4      int
#define u4_V850_PSW_ID  32      /* PSW */
#define u4g_gcgetv850psw()      PSW
#define DI()    atomic { PSW = PSW |  u4_V850_PSW_ID; INT = _pid }
#define EI()    atomic { PSW = PSW & ~u4_V850_PSW_ID; INT = -1 }
#define glint_di(scope)  {  u4 u4t_psw##scope;                          \
            u4t_psw##scope = u4g_gcgetv850psw();      \
            DI()
#define glint_ei(scope)        \
            if                      \
            :: (u4t_psw##scope & u4_V850_PSW_ID) == 0 -> EI()          \
            :: else                 \
            fi  }
int PSW = 0, INT = -1,  cnt = 0;
active [2] proctype task() provided (INT == -1 || INT == _pid)
{       glint_di(1);
            glint_di(2);
                cnt++;
                assert(cnt == 1);
                cnt--;
            glint_ei(2);
            assert(INT != -1);
        glint_ei(1);
        timeout ->
        assert(cnt == 0 && INT == -1)
}
```

Figure A.9-1. Spin Model of Recursive Interrupt Maski

A.9.2 Accelerator Pedal Position Learning

Purpose: Verify correctness of the released accelerator pedal position learning algorithm and the computed pedal position offset.

System: The pedal position module uses two sensors to compute a learned value for the released pedal position (i.e., with no pressure placed on the accelerator pedal). The learned zero position is used to determine the offset for the pedal position when the accelerator pedal is pressed. This offset is one of the key inputs to the throttle control algorithm.

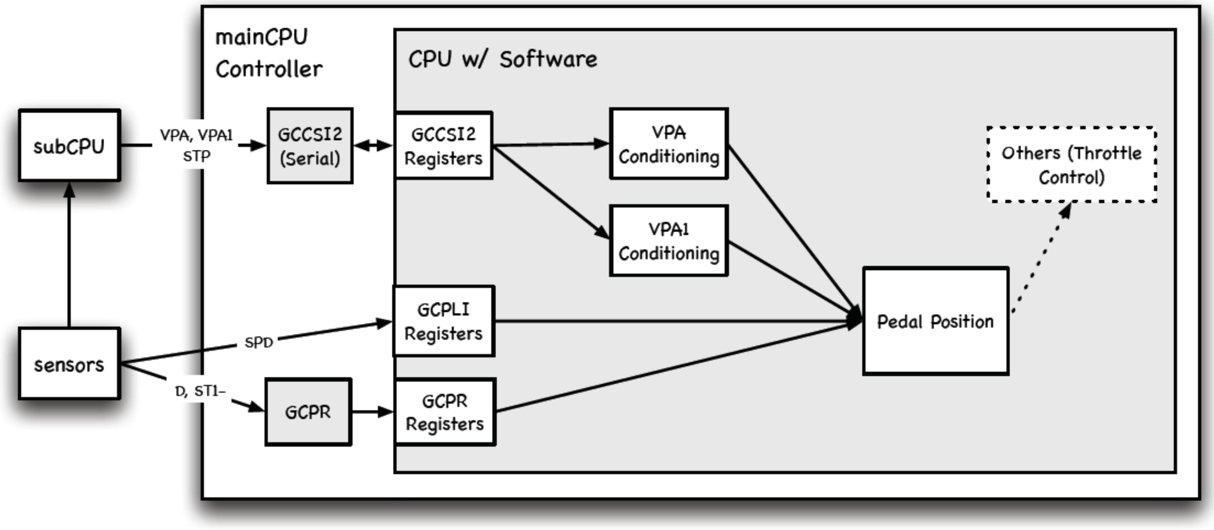


Figure A.9-2. Pedal Position System Context

Inputs: The primary inputs to the pedal position logic model are:

1. The two sensor values for pedal position (VPA and VPA1).
2. The vehicle speed (SPD).
3. The state of the brake-light, indirectly indicating the depression of the brake pedal (ST1, STP).

Outputs: The output of the pedal position logic model is the offset of the current accelerator pedal position from the learned value.

Model Description: The key features modeled and verified are based on a description of the intended working of the pedal position learning algorithm provided by Toyota engineers, and tracked in the code.

The accelerator pedal position is tracked as a voltage that can range from approximately 0.8 Volts (corresponding to an angle of 20 degrees) to 3.12 Volts (58 degrees). Three different ranges of pedal position changes are relevant to the learning algorithm.

- A change in value of less than 0.0388 Volt (small).
- A change in value between 0.0388 and 0.2 V (medium).
- A change in value larger than 0.2 V (large).

Similarly, three ranges of vehicle speed can influence the computation:

- A vehicle speed greater than 15 km/h (fast).
- A vehicle speed less than 3km/h (slow).
- Stopped.

The use of the brake pedal is modeled as the state of a Boolean variable, taking the value *true* or *false*.

Separately, the software determines based on the duration of input signal variations, whether or not the current pedal position should be considered released (*closed*).

The value of the released accelerator pedal position should change rarely, and only relatively small variations are anticipated. Causes for variation can be minor deformation of the pedal itself, the physical replacement of one pedal assembly for another by a mechanic, small variations in voltage output from sensors, etc. The most recently learned released accelerator pedal position is stored in SRAM.

There are various constraints imposed on updates of the learned value of the released accelerator position.

a. The learned value is updated only if the vehicle speed has been 15 km/h or more and the pedal sensor has reached 0.4 Volts or more, and the current vehicle speed is 3 km/h or less, the pedal sensor value is 0.2V or less, the brake switch light is on, and the pedal sensor reading has been stable for two seconds or more.

b. If (under these circumstances) the pedal closed position change in value is less than 0.0388 Volt (small), the learned value is updated, but each single update is restricted to a maximum of 0.01952 Volt (corresponding to a difference of 0.488 degrees).

c. If the change in value is larger than 0.0388 Volt, but less than 0.2 Volt (medium) then the learned value is not updated, and learning is suspended until the vehicle comes to a stop.

d. If the change in value exceeds 0.2 Volt (which Toyota said should physically be impossible) then learning is suspended for the remainder of the trip (i.e., until the ignition is turned off).

e. Variations of the value stored in SRAM are limited to a change of maximally 0.04 Volt per trip.

When the constraints c or d take effect, no record of this limit is recorded in the event data recorder, so it is not discoverable afterwards, nor is it know how frequently these events may occur.

One way in which the accelerator pedal released position could suddenly change is when a sensor spring that is used in the accelerator pedal assembly breaks. Toyota states that the maximum change in measured pedal position in this case will be less than 0.152 Volt (i.e., below the 0.2 Volt limit from constraint d above).

One scenario of interest in the consideration of the accelerator pedal learning algorithm is one in which the released accelerator pedal position decreases by a value that is larger than 0.0388 Volt, e.g. by 0.2 Volt. A value of 0.2 Volt would normally correspond to a 5 degree change in pedal position. In a scenario when learning is suspended (by constraints c and/or d) the new value would not be used, even if it were the correct new released pedal position. This would mean that even with the pedal in the released position the software would cause a pedal angle of 5 degrees to be used in the computation throttle angle module. With the pedal depressed, similarly values 5 degrees higher than correct would continue to be used.

> If the full range of the accelerator pedal position is 38 degrees (ranging from a stated minimum of 20 degrees and a maximum of 58 degrees) then a 5 degree anomaly corresponds to 13% of that range, which would indeed be noticeable to the driver.

The software team constructed various logic models of the pedal learning algorithm.

1. A first model is a direct representation of the pedal learning algorithm as document, and recorded in a PowerPoint summary provided by Toyota.
2. A second model is based on a Simulink/Stateflow model built by colleagues from the NASA Ames Research Center, part of the software study team.
3. A third model included the source code of the actual pedal learning algorithm, using the Spin model checker as a driver to exercise the code through relevant scenarios.

The first two models are limited in their value by being based on an interpretation of the working of the accelerator pedal learning algorithm as it was explained to the team by Toyota engineers and as documented in the PowerPoint summary provided. The details of the algorithm are sufficiently subtle that limited reliance on this indirectly obtained evidence can be placed.

The third model has the potential to be the most accurate, since it includes the actual source code with all its intricacies and details represented. Here, though, the problem encountered is limited ability to model the relevant portion of the *vehicle environment* in sufficient detail. The software team derived as much information as possible from the infrastructure used in unit-test files used by Toyota in the development of the code. In the unit tests time series of larger number of parameters (approximately 35) are used, significantly exceeding the primary input signals that feature in the high-level descriptions of the algorithm (accelerator pedal sensor readings, brake light switch, vehicle speed, etc.). The pedal learning algorithm is designed to function over longer periods of time (e.g., to evaluate the stability of the sensor readings). The abstract representation used for these external influences impacted the validity of the results. This example vividly illustrates the importance of an accurate environment model for the vehicle, as it is often used in a sub-system testing or system simulation for complex systems. No such model was available to the software team, and the construction of an accurate environment model could have meant months of detailed interaction with domain experts, which the team did not consider to be feasible within the constraints of the work. Therefore, this study is limited to an approximate model of the relevant input parameters.

Claims w/ Results: The software model correctness claim verified for each model was that it is impossible for the computed pedal position to be non-zero when the accelerator pedal is in the released position. Given the nature of the constraints on the pedal position learning algorithm, this claim can be shown to be violated. The pedal position can be shown to indicate non-zero even when the pedal is released. Result: Inconclusive. The model and claim could not be completely analyzed.

This pedal position learning issue influenced the development of testing done on vehicle hardware.

A.9.3 Sensor Input ADC (GCCSI2)

Purpose: Confirm that the processing of 10-bit and 12-bit analog-to-digital conversions is handled correctly in the multiprocessing environment. Accelerator pedal and throttle angle are input via the 10- and 12-bit ADC interfaces. Would corruption of these values, even if temporarily, impact the subsequently computed new demanded throttle angle?

System: Analog sensor data is feed to the Sub-CPU, is sampled and serialized in the Sub-CPU, passes over to the Main CPU via the serial interface, and is then distributed within a couple of tasks to a variety of modules. Tasks are initiated by interrupts and explicit OS activation. Delays in completion may cause ADC operations to overlap in time.

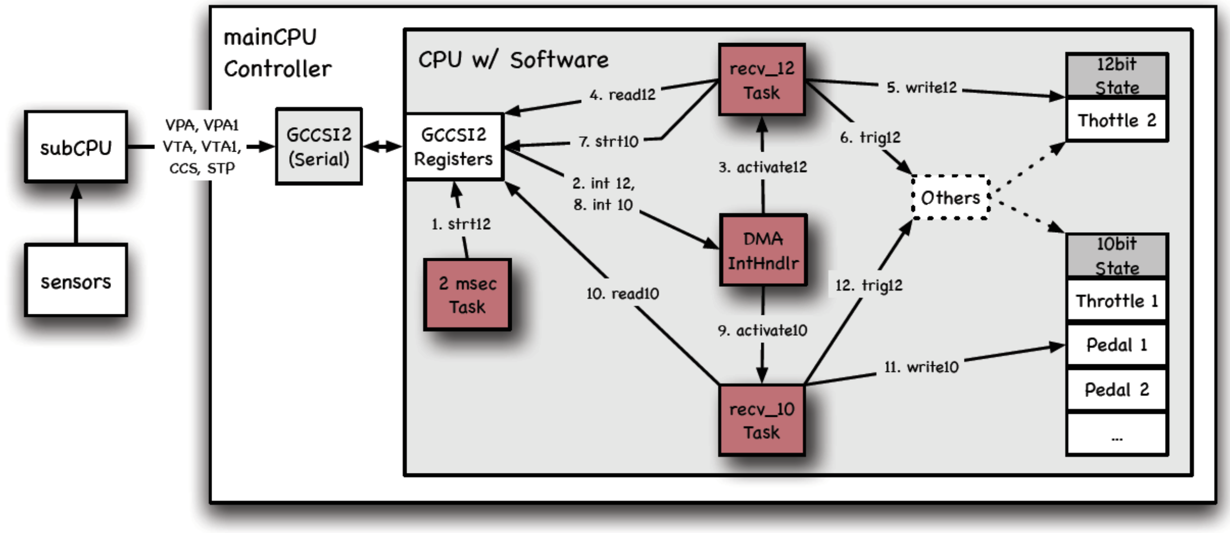


Figure A.9-3. Sensor Input ADC System Context with Software Collaborations

Figure A.9-3 shows an abstraction of the software collaboration among the ADC tasks. None of the error handling is shown and the order of the collaborations, denoted with the message numbering, is idealized. The logic model is designed to explore the overlap of operations on the GCCSI2 Registers owing to multiprocessing timing variations.

Inputs: The inputs for the sensor input ADC that are relevant to the throttle control are:

- Throttle Position A/B - The throttle position is sensed from two sensors at both a 10 and 12 bit value.
- Pedal Position A/B - The pedal position is measured by two sensors as two 10 bit values.
- Cruise Control Switch - The cruise control switch signal is sensed as a 10 bit value.

Within the logic model these specific inputs are abstracted to 10 bit input and 12 bit input. That is, the logic model is only designed to look for 10 bit and 12 bit issues in the GCCSI2 Register commanding - not issues in the specifics of throttle position A or B, for example.

Outputs: The outputs from the sensed input ADC is state date written in software memory. The state data is then accessed by a variety of other software modules. In the logic model this output data is not relevant.

Model Description: The logic model for the sensor input ADC consists of six Promela *proctypes*. The GCCSI2 peripheral IC computation is captured with a single prototype. The other five Promela *proctypes* are coordinated using a priority scheme that is analogous to the ADC-related OSEK tasks in the Toyota code. These five proctypes are further coordinated using the Promela 'tasking' library which ensures that the periodic timing of the *OSEK* tasks, typically keyed off of a 2ms task, is faithfully represented in the logic model. The GCCSI2 peripheral IC *proctype* is fully asynchronous relative to these five other *proctypes*.

A significant compilation parameter in the logic model is whether or not the proctypes have sufficient margin to complete within each and every cycle. When margin does not exist, any proctype could be delayed beyond a cycle and thus overlap with other proctypes that have started on that next cycle.

The logic model contains Promela for two safety properties. One check ensures that the enable/disable logic for the GCCSI2 peripheral IC occurs in pairs. The other check ensures that 12-bit data is read when in 12 bit mode and similarly for 10 bit data. These two safety properties implement the correctness claims for this Sensor Input ADC model.

Claims w/ Results: The claims are designed to expose overlaps in the commanding of the GCCSI2 registers.

1. Confusion between 10 and 12 bit reads - The GCCSI2 is never read for 12 bit data when 10-bit data is being supplied and analogously for reading 10 bit data.

   Result: Verified when there is margin. Fails when there is no margin.

2. Enable/Disable skew - The GCCSI2 is never disabled or enabled twice in a row -- the correct behavior to an alternating sequence for enable and disable.

   Result: Verified when there is margin. Fails when there is no margin.

This software model shows that when margin exists the 10 and 12 bit reads cannot overlap. However, if there is no margin, then there is possible overlap. The 'no margin' case was discussed at length with Toyota engineers. Toyota stated that 1) an overlap between the 10 and 12 bit reads will have no negative consequences and 2) the delays that are required to produce an overlap are unrealistically large.

A.9.4 Motor Drive IC

Purpose: Confirm that the MotorDriveIC internal logic drives the throttle motor's H-Bridge transistors correctly. One cause of UA is that the MotorDriveIC opens the throttle plate when the inputs do not demand an open throttle plate. This logic model is designed to find this UA condition. This model also checks for H-Bridge transistor configurations that lead to an electrical short when, like UA, those configurations are not warranted by the inputs.

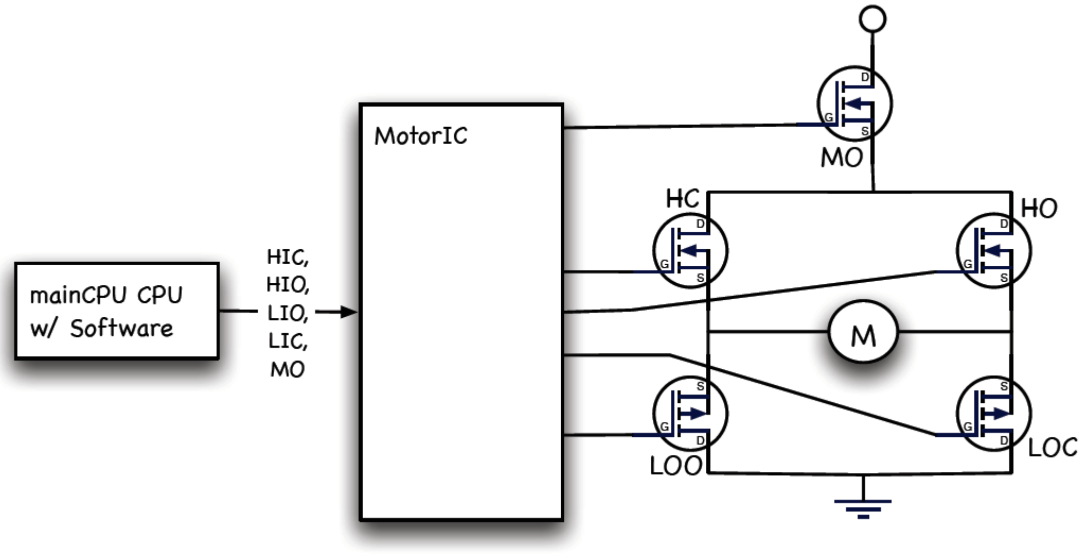


Figure A.9-4. Motor Drive IC System Context

System: The MotorDriveIC system context is illustrated in Figure A.9-4. The MotorDriveIC outputs a set of binary signals that open or close high power transistors in the throttle motor's H-Bridge. Combinations of transistors drive the throttle plate either opened or closed; some combinations, if they occur, can produce an electrical short. Note that driving the throttle closed is required because the throttle plate's un-actuated position is determined by counter balancing springs. The neutral spring position produces an angle that is larger than required to idle the engine.

The MotorDriveIC internal logic is Toyota IP. The functionality is non-trivial in that there are feedback loops, time delays and internal memory (latches). A logic model of this functionality will ascertain if the desired functionality is achieved in all cases, for all possible inputs.

Inputs: The Main CPU ASIC produces binary signals as inputs to the MotorDriveIC. The inputs to the MotorDriveIC are:

- HIC (boolean) - The input driving the HC H-Bridge transistor.

- HIO (boolean) - The input driving the HO H-Bridge transistor.
- LIO (boolean) - The input driving the LOO H-Bridge transistor.
- LIC (boolean) - The input driving the LOC H-Bridge transistor.
- MO (boolean) - The input driving the MO H-Bridge transistor.

The logic model independently randomizes each of these inputs thereby exploring the entire input space. Clearly many combinations of these inputs may not be produced by the Main CPU software but these combinations are valid when exploring the MotorDriveIC correct behaviors.

Outputs: The outputs from the MotorDriveIC are binary signals that drive five H-Bridge transistors. These outputs are the H-Bridge transistors labeled HC, LOO, LOC, HO and MO in Figure A.9-4A.9-4. There are a number of important combinations of these outputs (assuming MO is 'closed'):

- HC+LOC open the throttle plate.
- HO+LOO close the throttle plate.
- HC+LOO or HO+LOC create an electrical short.

The logic model checks each of these constraints.

Model Description: The logic model for the Motor Drive IC is derived from the digital circuit schematic for this IC. The logic model uses two fully asynchronous Promela *proctypes*. One *proctype* models the binary input signals which can change independently of the IC logic. These inputs signals are fully randomized within this proctype. The other *proctype* models the IC logic and produces the IC outputs. This later *proctype* updates internal state, captures feedback in the IC and allows the outputs to settle. It also models over current and over temperature behaviors.

This logic model has a single safety property to check if the SR-latch avoids the unstable state (S=1, R=1).

This logic model has a number of *never* claims. A couple of the *never* claims are used to check the correctness of the model itself. Besides these claims, the primary claim checks that the throttle motor is never eventually always open unless the inputs warrant a wide open throttle. The logic model includes the logic that relates the pairs of outputs that open and that close the throttle.

Claims w/ Results: For verification of the MotorDriveIC, correctness claims are formulated. These claims ensure that dangerous output combinations are avoided, that the outputs are consistent with the inputs, and that various internal constraints are not violated. The claims verified are:

1. Verify that the throttle plate is never eventually always wide-open when the inputs are not always demanding wide open.

   Result: Verified. The claim made upon the model being checked is not violated on any path checked in the model.

2. Verify that an electrical short never occurs unless the inputs demand that an electrical short be produced.

   Result: Verified. Note that this model does not include details of the Toyota IC that are designed to prevent an electrical short circuit, even if the inputs demand one.

3. Verify the all SR-latches are never in their unstable state (S=1, R=1 simultaneously).

   Result: Not verified. When an SR-latch is unstable the output is undefined. Questions to Toyota engineers on this claim failure resulted in an explanation that the SR-latches from the design documentation are in fact S-latches which do not have an unstable state. Thus this claim is extraneous.

The MotorDrive IC was shown to satisfy claims 1 and 2. Claim 3 was shown to be unimportant. This model effectively rules out an error in the MotorDrive IC as a cause for UA.

This software model indicates the Motor Drive IC outputs can only lead to a wide open throttle when the Motor Drive IC inputs warrant a wide open throttle.

A.9.5 Port Register Inputs (GCPR)

Purpose: Confirm that the processing of the port register inputs is faithful to the sensed input values. This ‘GCPR’ functionality, from sensed input value to concluded state, involves a multiple asynchronous tasks. This models checks if multiprocessing errors are avoided.

System: A number of inputs are provided through the port registers. These inputs are read periodically by a recurring CPU software task, processed and then written to software state. That state is then used in subsequent throttle computations.

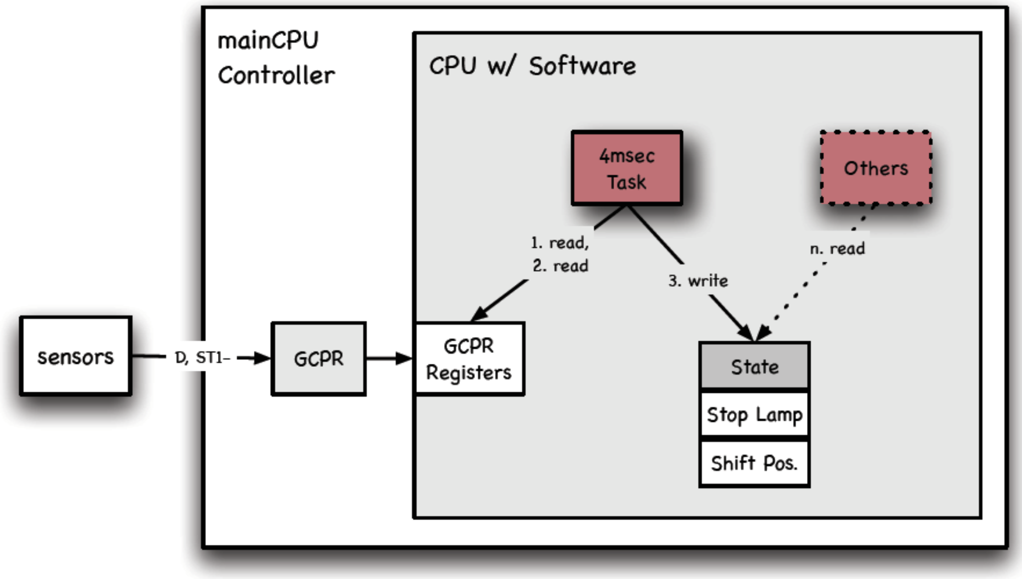


Figure A.9-5. Port Register Inputs (GCPR) System Context

The periodic task reads the port registers twice, nominally for 'noise canceling.' While these two reads occur on a single line of C source code, the multiprocessing nature of the throttle control software allows for the possibility that a significant delay can occur between the reads. Furthermore, because the port registers reflect the sensors in the environment, the values in the port registers can change asynchronously relative to the CPU software. This leads to the possibility that the two reads produce different values.

Inputs: The logic model uses two sensor values that are input through the port registers:

1. Shift Position Switch Sensor (boolean) - Indicates if the transmission is in 'drive'
2. Stop Lamp Switch Sensor (boolean) - Indicates if the stop lamp is on and thus, indirectly, if the brake pedal is depressed.

Within the logic model these two inputs are independently randomized to explore the entire inputs space. Note that there are no dependencies between these two variables, either in the GCPR code or in the logic model.

Outputs: The outputs are the shift position state and the stop lamp state. These are the states used in all subsequent throttle processing by the CPU software.

Model Description: The logic model for the Port Register Inputs is a small model consisting of two Promela *proctypes*. The first *proctype* represents the asynchronous GCPR peripheral IC. This *proctype* simply randomizes the possible inputs. The second *proctype* represents the *OSEK* task in the Toyota code and serves to sample the port register data. The data is read twice nominally for noise canceling purposes. The logic model checks for a race condition between this double read and the GCPR peripheral IC *proctype*.

This logic model has a number of claims. Some of the *never* claims are used to check the correctness of the model itself. The primary claim ensures that the output and the input never "*eventually always*" differ.

Claim w/ Results: For verification of the 'Port Register Inputs,' a correctness claim is formulated to ensure that the outputs are consistent with the inputs. The computation of the outputs involves a handful of steps that may be susceptible to corruption owing to asynchronous task race conditions.

1. The shift position state will never be eventually always 'open' unless the shift position switch sensor is always 'open.' Due to the symmetry in the GCPR code and this logic model, this claim also applies to the brake light switch.

   Result: Verified. The claim made upon the model being checked is not violated on any path checked in the model.

The claim was verified. This effectively eliminates corruption of the *Shift Position* and *Brake Light* switch through the GCPR interface as a possible cause of UA.

A.9.6 PWM Functionality (GCPLS)

Purpose: Confirm that the PWM (Pulse Width Modulation) functionality reliably modulates the motor IC transistors based on the desired PWM duty cycle.

System: The PWM functionality is achieved with a number of software tasks that are activated based upon hardcoded delays. Specifically the final output to the GPCLS registers is initiated after a ~800 microsecond phase offset within the 2 msec task. Figure A.9-6 illustrates the PWM system.

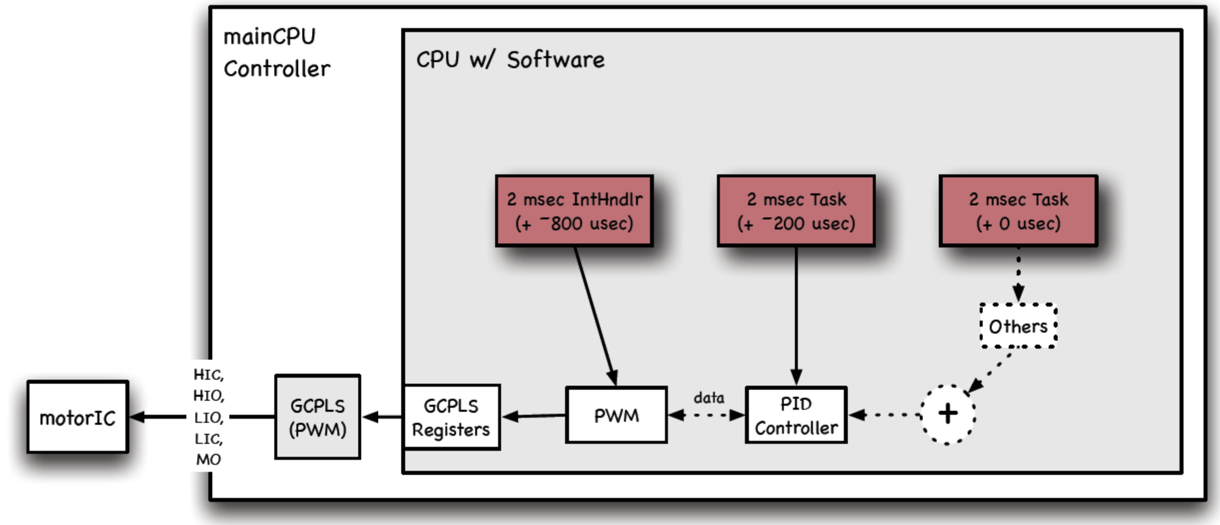


Figure A.9-6. PWM Functionality System Context with CPU Tasks

Data is shared between the 'PID controller', which produces the 'duty cycle' for the throttle motor, and PWM which smoothly handles the throttle motor motion using timed actuation of the H-Bridge transistors. The logic model explores consequences of arbitrary delays in any of the periodic 2 msec task computations.

Inputs: The PWM functionality is invoked as part of a periodic software task with inputs produced by the 'PID Controller' module. The primary input is the desired PWM duty cycle and its direction (positive/open or negative/close). The inputs to the PWM module that are used in the logic model are:

- Activate (boolean) - Determines if the motor IC should be used or not.
- Reset (boolean) - Determines if the motor IC should be reset.
- Duty Cycle (enumeration) - Determines the fraction of 2 ms during which the throttle motor, though the MotorDriveIC, should be powered. A fraction of +100% implies 'wide open throttle' and a fraction of -100% implies a 'fully closed throttle'. Anything above approximately 88% is rounded up to %100.
- Duty Cycle Direction (boolean) - Determines the direction of throttle motor rotation. '1' implies 'open' and '0' implies 'closed'.

Outputs: The outputs from the 'PWM Modes' computation are binary signals that serve as inputs to the MotorDriveIC. Thus, these PWM outputs are directly responsible for opening and closing the throttle motor. These outputs are HIC, LIO, HIO and LIC and are shown in Figure

A.9-4. Because this PWM outputs drive the H-Bridge transistors there are valid and invalid combinations.

Model Description: The logic model for the PWM Functionality consists of three Promela *proctypes*. One *proctype* represents the GCPLS timer peripheral IC and operates asynchronously to the other *proctypes*. Another *proctype* in this model represents an OSEK interrupt handler. The final *proctype* represents a periodic task that is possibly inhibited by the interrupt handler and which computes the desired PWM model.

The *proctype* for the GCPLS timer peripheral IC sets binary values based on an 'on time' and an 'off time'. This *proctype* uses the Promela 'task' library to define an alarm which runs repeatedly at each clock tick. When it runs the *proctype* compares the current tick with the 'on time' and the 'off time' to determine if the binary value should be set. The binary values in this *proctype* represent the on/off state of the throttle motor transistors.

The *proctype* for the interrupt handler uses the current PWM mode and duty cycle to compute the on and off times for the four H-Bridge transistors in the Motor Drive IC. This *proctype* then outputs the computed times to the GCPLS *proctype*. This computation closely mimics the Toyota code with the introduction of delays and with the handling of the PWM modes. This *proctype* nominally runs every [redacted] with a phase offset of [redacted]. However, the Promela scheduler can introduce arbitrary delays in this *proctype*'s execution.

The final *proctype* computes the PWM mode based on the duty cycle. This *proctype* fully randomizes its input, which is the duty cycle, and then derives the mode and related state. This *proctype* runs every [redacted] with a phase offset of [redacted] but, because of scheduling variations, it can run with an arbitrary delay even to the point of overlapping the interrupt handler.

The safety property in this logic model represents an invalid combination of throttle motor transistors. The property ensures that only two transistors can be power at one time.
The *never* claims in this logic model extend the safety property based on tighter constraints on the valid combination of transistors and the relationship to required PWM duty cycle. The primary claims ensures that the throttle is never eventually always wide open unless the requested duty cycle is at [redacted]

Claims w/ Results: For verification of the ‘PWM Modes’ the formulated correctness claims are based on valid and invalid combinations of outputs. Valid combinations are those that lead to

1. Three outputs are never simultaneously binary ‘1’ (read as ‘on’ or ‘closed’).

   Result: Not verified. The claim made on the model being checked is violated on at least one path checked in the model.

2. Two outputs that lead to an H-Bridge electrical short are never simultaneously binary '1'. It is considered an error for a short to be commanded by the PWM module even though the MotorDriveIC may prevent an actual electrical short.

   Result: Not verified. The claim made upon the model being checked is violated on at least one path checked in the model.

3. The outputs that open the throttle plate is never eventually always open (leading to a wide open throttle plate) unless the requested duty cycle is [redacted]

   Result: Verified. The claim made upon the model being checked is not violated on any path checked in the model.

The logic model failed to verify claim #2. The failure occurs for the following conditions:

1. The PWM mode is PWMPLS with a duty cycle near but less than [redacted].
2. The [redacted] phased computation is delayed by [redacted].
3. The PWM mode becomes PWMMNS.

These conditions were discussed with Toyota engineers with the [redacted] delay garnering the most attention. At first Toyota did not consider any delay of the [redacted] phased computation as credible however, after undertaking numerous measurements, Toyota observed delays as large as [redacted]. Subsequent to this recognition, Toyota computed an absolute worst case delay. This worst case is based on all possible relevant delays conspiring and was computed to be [redacted].

Further analysis of the MotorDriveIC shows that a software-induced electrical short is prevented by the hardware. Thus, even if claim #2 could occur, there would be no adverse effect on the vehicle. The improper software command is filtered by the hardware until it subsides at which point a proper H-Bridge configuration is re-established to drive the motor open or closed.

The Pulse Width Modulation (PWM) code will only lead to a wide open throttle when the requested duty cycle is +100%.

A.10 Software Algorithm Design Analysis Using Mathworks Models

A.10.1 Modeling Effort Overview

In the Toyota 2005 Camry, the critical control loops that are related to potential UA are implemented in software that executes in the electronic throttle control system (ETCS-i). In order to analyze the control loops and to support hardware testing scenario development, an in-depth understanding of Toyota's throttle control system as implemented in software is necessary. Matlab, Simulink, and Stateflow were used to model the control system at a level of abstraction suitable for control system insight and analysis. Using Toyota's source code and specifications, along with consultation of Toyota systems experts, executable models of the ETCS-i were constructed through an iterative process of reverse engineering. Toyota was very co-operative in this process, providing engineers with expertise in successive subsystems who stayed in Torrance or San Jose on average two weeks at a time.

These models were provided to the entire NASA team and served as a common basis of analysis for both hardware and software teams. After each major subsystem was modeled, the software and hardware teams met face-to-face in order to share information and develop testing strategies. The software team made extensive use of both Mathwork's test tools as well as NASA-developed tools to generate and simulate test suites for a wide range of driving scenarios involving both nominal and off-nominal situations, such as sensor faults. Simulink provides a modeling environment for data-flow centric control, while Stateflow provides a modeling environment for complex discreet logic. Given the Mathworks' tools position as leading mathematical computing software for engineers and scientists in the aerospace and automotive industry, these tools were well-situated to support the Toyota analysis.

The goal of the modeling exercise was to develop an integrated system model of the ETCS-i. To aid in the testing of the various subsystems it was decided to split the model into to an integrated portion, the Idle Speed Control (ISC), and the idle on fuel cut system. The reason to split up the model was because of control feedback loops that required a plant model. For instance, the ISC required a plant model of the engine to receive accurate feedback for control purposes. The development of these plant models was outside of the scope of this investigation. Table A.10-1 describes statistics for 05 Camry L4 Matlab Modeling.

Table A.10-1. Model Statistics

Model	Functional Blocks	Stateflow Blocks	Transitions	States	NASA Man-Hrs	Toyota Man-Hrs
Integrated Model w/Cruise Control	1038	17	1088	623	200	100
Cruise Control	229	8	365	222	70	50
Idle Speed Control (ISC)	935	42	615	448	120	80
Idle On Fuel Cut	76	4	51	26	4	2

Based on a preliminary review of the software specifications and discussions with the Toyota engineers, it appears that the Vehicle Stability Control (VSC) and Transmission software doesn't increase the command to the throttle valve IC, therefore models of the VSC and the Transmission software were not developed.

Considering the complex and interactive nature of the models that were developed, it was necessary to vary the fidelity of each of the models to match the testing needs and scenario development process. The models were developed to three levels of fidelity that are described in the chart below. In addition, a software build in the native Greenhills compilation and microprocessor emulation environment was developed. This environment reached sufficient maturity to map inputs leading to diagnostic codes.

Note: The following figures of the Camry MY2005 models illustrate the complexity and depth of the modeling effort. They are not “readable” by intent to protect the details of the Toyota design.

A.10.2 Model Development

The models were developed in three phases. Each phase varied the level of fidelity. Modeling in different phases allowed for efficient modeling, testing, and production of usable results for a highly complex integrated system.

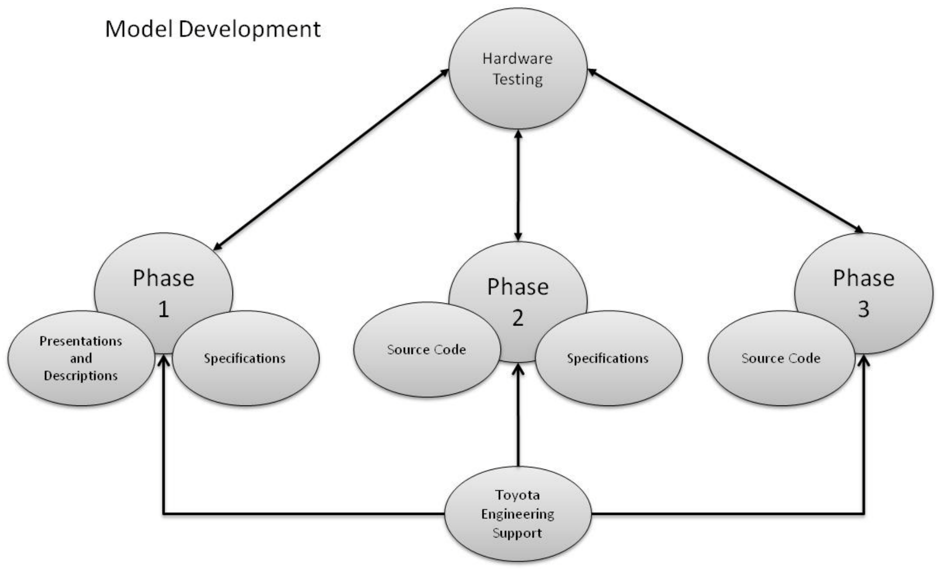


Figure A.10-1. Model Development and Fidelity

A.10.3 Phase 1

These models are based on presentations or diagrams directly from the Toyota engineers as well as minimal use of specifications. These are relatively low fidelity models and produced results that yield a minimal understanding of the functions. Because the Phase 1 models are mostly used to develop an understanding these models do not always use the same variable names and are primarily concerned with getting the logic correct. These models were verified by comparison with expected results from the Toyota engineers.

The following functions are of Phase 1 fidelity:

A.10.3.1Safing Functions

In the first phase the primary interest was the modules that increase the throttle angle. For this phase a simple model of the safing function block would suffice. A Stateflow model was developed from a Toyota provided list of diagnostic codes and resulting fail-safe responses. Developing a higher fidelity model would prove useful in providing a better understanding of the fail-safe actions and how and when they are performed.

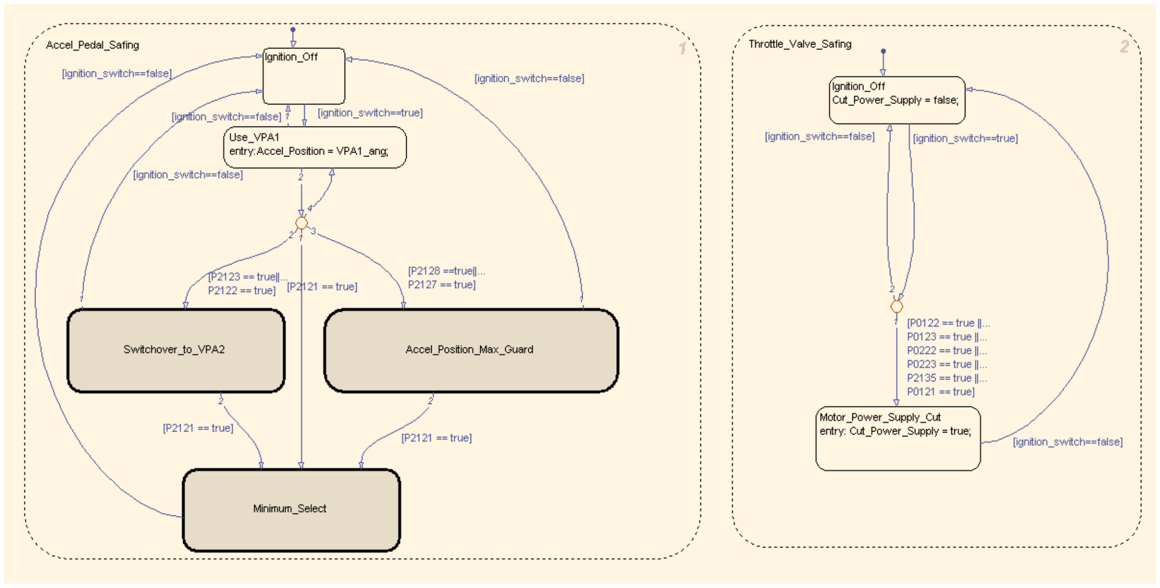


Figure A.10-2. Safing Functions Model

A.10.3.2Accelerator to Throttle Demand

Toyota described the function of the following module as a lookup table and therefore it was determined that no further increase in fidelity was necessary.

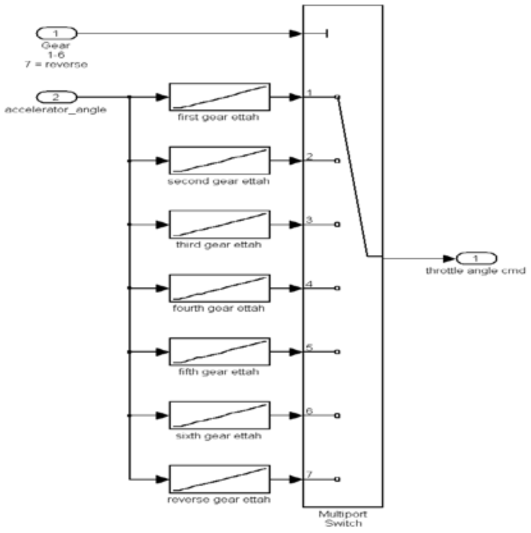


Figure A.10-3. Throttle Demand Model

A.10.4 Phase 2

The models were primarily based on interpretations of the specification documents related for software functions provided by Toyota to Denso. These models are at a higher fidelity then the Phase 1 models, however, they still model the expected outcome and, if the specifications do not match the source code, then the Phase 2 model might contain discrepancies between the hardware and the model.

The following functions are of Phase 2 fidelity:

A.10.4.1 Cruise Control

The cruise control is implemented through a single voltage input that is manipulated by the driver through a switch. Depending on the input from the driver, the switch will select a resistance that is interpreted by the cruise control logic to set the state of the cruise control. The input voltages are received by a module that interprets which position the switch is in. The next module uses the position determination and other inputs to decide which state the cruise control system is in. Depending on the determined state, a control module is used to complete the actual calculations required to send a command to the vehicle. To accomplish these calculations there is a module within the cruise control that calculates the vehicle speed. Various states and voltages within and outside the cruise control are monitored through diagnostics. These diagnostic modules then feed into the control module to be used in the auto cancel operations.

Note: The following figures of the Camry MY2005 models illustrate the complexity and depth of the modeling effort. They are not “readable” by intent to protect the details of the Toyota design.

[redacted]

[redacted]

A.10.4.2 Idle Speed Control (ISC)

The ISC incorporates modules that provide throttle contribution to maintain idle and compensate for conditions like creep control, increases in oil temperature, variable valve timing, alternator loads, air conditioner loads, catalyst temperature, idle while moving, stall prevention, fuel cut, variations in the throttle valve assembly, purging, power steering, startup/ignition, and engine temperature. Three modules are used to complete the learning algorithm calculations that compensate for manufacturing variances and throttle deposits that accumulate over time. There are a series of modules that then take the outputs from all of the contributing blocks and add them together to calculate a final throttle angle demand from the ISC system.

Note: The following figures of the Camry MY2005 models illustrate the complexity and depth of the modeling effort. They are not "readable" by intent to protect the details of the Toyota design.

[redacted]

A.10.4.3Idle On Fuel Cut

As depicted below, the fuel cut function has a module that determines the rpm at which to return from fuel cut based on a set of inputs from other systems. The next module makes a determination of whether to implement fuel cut return. After the return determination is made, the next module calculates the rpm to execute fuel cut. The final module makes a determination of whether to implement fuel cut based on the inputs from the rpm calculation and whether or not fuel cut return is being implemented.

[redacted]

A.10.4.4 Diagnostics

Diagnostics are a major component of the functionality of the ETCS-i. This component is naturally of high interest to the Toyota ETCS-i study. The model of the diagnostics for pedal, throttle, and power, were utilized in the hardware testing to direct the testing toward areas which would not send the vehicle into a fail-safe state. In other words, sensor anomalies that resulted in diagnostics leading to a fail-safe state precluded UA.

The diagnostic model is a collection of a series of smaller models of each of the pedal and throttle DTCs.

The model was developed using a spreadsheet containing a description of the logic, which was provided initially to the California Air Regulations Board (CARB). After finding a number of errors in the spreadsheet, the model evolved to use the specifications, source code, and most importantly, the insight and knowledge of the Toyota diagnostic experts.

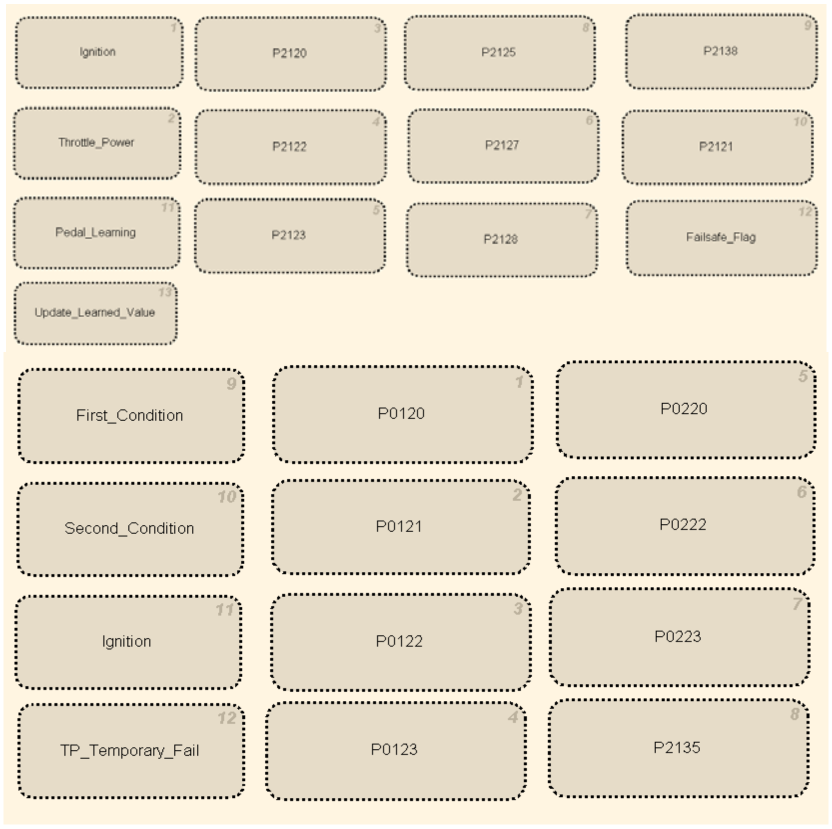


Figure A.10-8. DTC and Diagnostic Model

PID - The PID manages the difference between the commanded throttle angle and the sensed throttle angle and determines what duty cycle command will minimize the difference between the two.

Learning Functions - The learning functions calibrate the control software to accept and adjust to pedal and throttle sensor inputs which are only slightly off the nominal path. This function provides optimal driving performance that remains constant.

The learning functions were of high interest to the NASA team, because of the learning function’s ability to mask failures and to change the performance of the vehicle as measured against absolute sensor values, by virtue of changing the zero bias. The learning function models evolved over the progression of the study and were a valuable tool in the testing process.

Because of the proprietary nature of phase 2 type models, lower level diagrams cannot be shown. Below is a higher-level snapshot of the inputs and outputs of a phase 2 model of the learning algorithm.

[redacted]

A.10.5 Phase 3

In order to provide the highest fidelity possible within a Mathworks simulation environment, the source code for the ECU used in the Toyota MY05 L4 Camry was compiled on the Linux environment. This enabled engineers to generate S-functions that can be loaded into Matlab or Simulink blocks. Using the source code directly improves the accuracy of the simulations and help the engineers validate models based on schematics and descriptions. Because the S-functions are within the Simulink environment, they can be integrated into the same testing framework as the models in the previous phases replacing blocks designed on schematics. The compiled code also increased execution time allowing more exploration of the testing scenarios space.

The Toyota source code is complex and has many dependencies that make full-scale simulation outside its native hardware environment difficult. Even within the Greenhills compilation and debug environment, the practice of Toyota and Denso has been to only use a software simulation environment for “one-shot” unit testing, i.e., one input vector yielding one output vector. Any further testing beyond the unit level was done by Toyota and Denso on hardware platforms with integrated software loads.

The compilation and linking into s-functions were finally successful. However, the full incorporation into the Mathworks simulation environment did not reach maturity in time for Phase 3 models to be incorporated into the current model used in testing. This could be continued as forward work.

A.11 Mathworks Model Scope and Functional Description

The scope of the integrated model was to provide coverage on model three major contributors to the throttle control, Accelerator Pedal, Cruise Control, and the ISC. These three major contributors each have the potential to increase the throttle a significant amount.

Figure A.11-1 shows an overall simplified functional view of the ETCS-i functions that have been incorporated in the integrated model. Description, verification, analysis and testing, and results of the models of each of the functions are described in the following section.

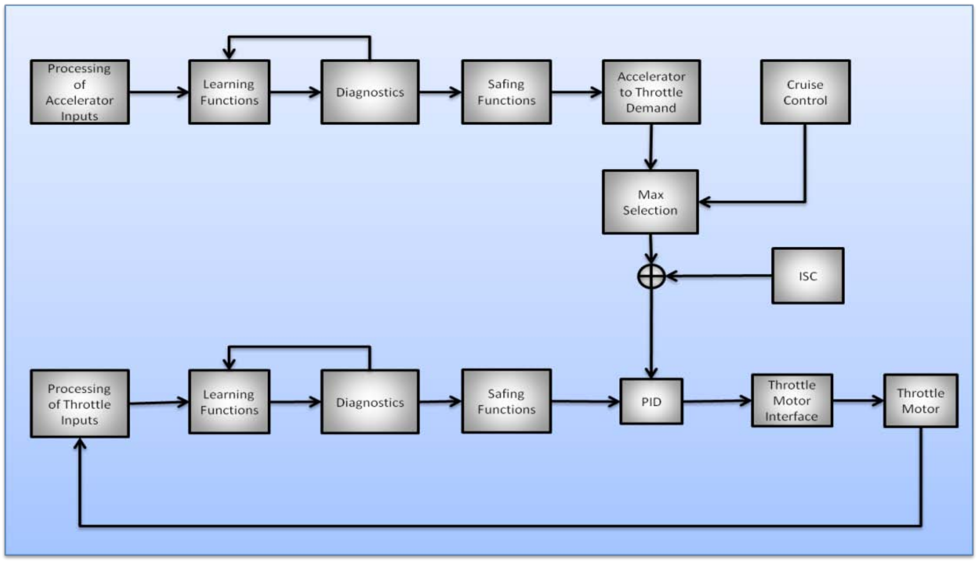


Figure A.11-1. Architecture of Modeled Throttle Control Functions

A.11.1 Accelerator Pedal Control

The pedal controls the throttle by measuring the pedal command angle and comparing it to the learned pedal released value. Using the command and the learned pedal release value, pedal diagnostics are performed. When a fail-safe flag is sent from the pedal diagnostic algorithms, certain fail-safe responses are executed to limit the throttle opening. The pedal command angle, after going through the diagnostic and fail-safe processing, is converted to a throttle commanded angle. The throttle command angle from the pedal input is compared to the throttle request from the cruise control system. The greater value of pedal throttle command and cruise control request is then sent to the PID controller. The PID then calculates a motor duty cycle command and this command is converted into ON-OFF commands to four transistors which control the flow of electricity to the throttle motor.

A.11.1.1Processing of Pedal Input Functions and Learning Functions

The pedal control's primary input comes from two pedal sensors, VPA1 and VPA2. VPA1 is used for primary control and VPA2 is used to check the validity of VPA1. VPA1 and VPA2 have an input range between 0 and 5 V and are normally offset from each other by 0.8 V. The nominal range is shown below in Figure A.11-2.

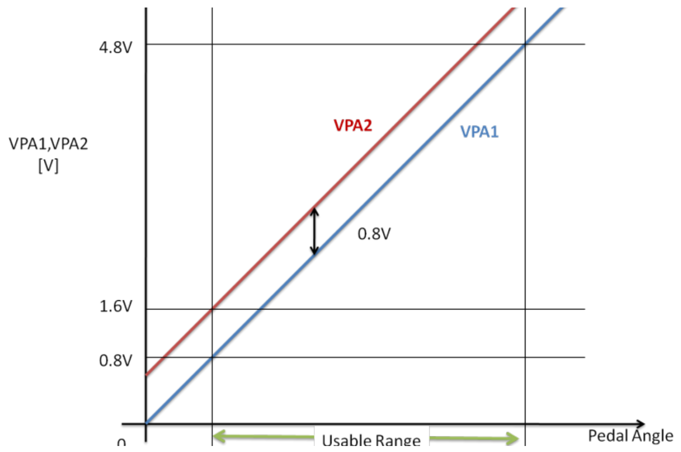


Figure A.11-2. Nominal Software Range for VPA1 and VPA2

A.11.1.2. Learning Functions and Released Position

To allow for recalibration of the zero bias during a trip, the pedal input goes through a preprocessing function, which recalibrates the pedal sensor input to allow for these input variations. The calibration process occurs anytime the pedal is determined to be released. [redacted]

Investigation of the software encompassing the learning algorithm concentrated on hardware failure sequences that result in the learning algorithm giving false indications of the pedal release value resulting in a true pedal release to be interpreted as a command.

A.11.1.2.1Pedal Learning Algorithm

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

If there is a fail-safe condition detected after learning is completed, the learning value is reset to the default value, as shown in Figure A.11-5.

A.11.1.3 Pedal Diagnostics

Based on individual pedal sensor and sensor to sensor correlation, checks are performed to determine the validity of the sensor data entering the CPU. To effectively understand and evaluate the range/area of valid or invalid values, the team used the model to generate plots or "maps" of the two pedal position sensors. The horizontal axis of these maps is VPA1 voltage, and the vertical axis is the VPA2 voltage. A sweep through all possible voltage relationships was performed. When a software diagnostic detected an invalid condition, it was noted on the map.

From this map, the valid acceptable VPA1 and VPA2 voltages were identified, and all regions where the software would detect invalid inputs were identified.

A second map was identified that allowed a wider range of valid values. This map is only applied when battery power is reapplied. After a valid fully released pedal position is learned, the map with the constrained or narrow range of valid values is applied.

The model's diagnostic maps were compared with the actual source code within the Greenhills Multi Environment as well as Camry hardware to ensure validity.

Six maps were generated for both the pedal diagnostics and pedal fail-safe detection. Three maps included the expanded valid range and three the narrow valid range. For both expanded and narrow threshold regions, a map was created for a ramp, step, and constant input sequence. The ramp, step, and constant input sequences were necessary to allow timing and sequence logic to function within the diagnostic detection code.

An example of two of the ramp sequences is illustrated below. This sequence starts and holds VPA1 and VPA2 at their nominal voltages of 0.8V and 1.6V, respectively, for 3 seconds, and then, in 2 seconds, linearly moves to the new VPA1 and VPA2 value, which it holds for 2 seconds. This sequence is performed for all new VPA1 and VPA2 values. The diagnostic and fail-safe detection maps are shown below in Figures A.11-6 and A.11-7 for this 2 second ramp sequence.

In Figure A.11-6, for every combination that produced a diagnostic code, it was marked with that diagnostic code. An operational range of values is identified starting in the lower left and continuing to the upper right. The lower left plots the pedal fully released voltages. The upper right plots pedal fully pressed voltages.

Figure A.11-7 is plotted in a similar manner. For every combination that produced a diagnostic code, it was marked with that diagnostic code. An operational range of values is identified starting in the lower left and continuing to the upper right. The lower left plots the pedal fully released voltages. The upper right plots pedal fully pressed voltages. The range of acceptable values not producing a diagnostic code is wider. This range is the acceptable range of values after battery voltage is reapplied, such as after maintenance.

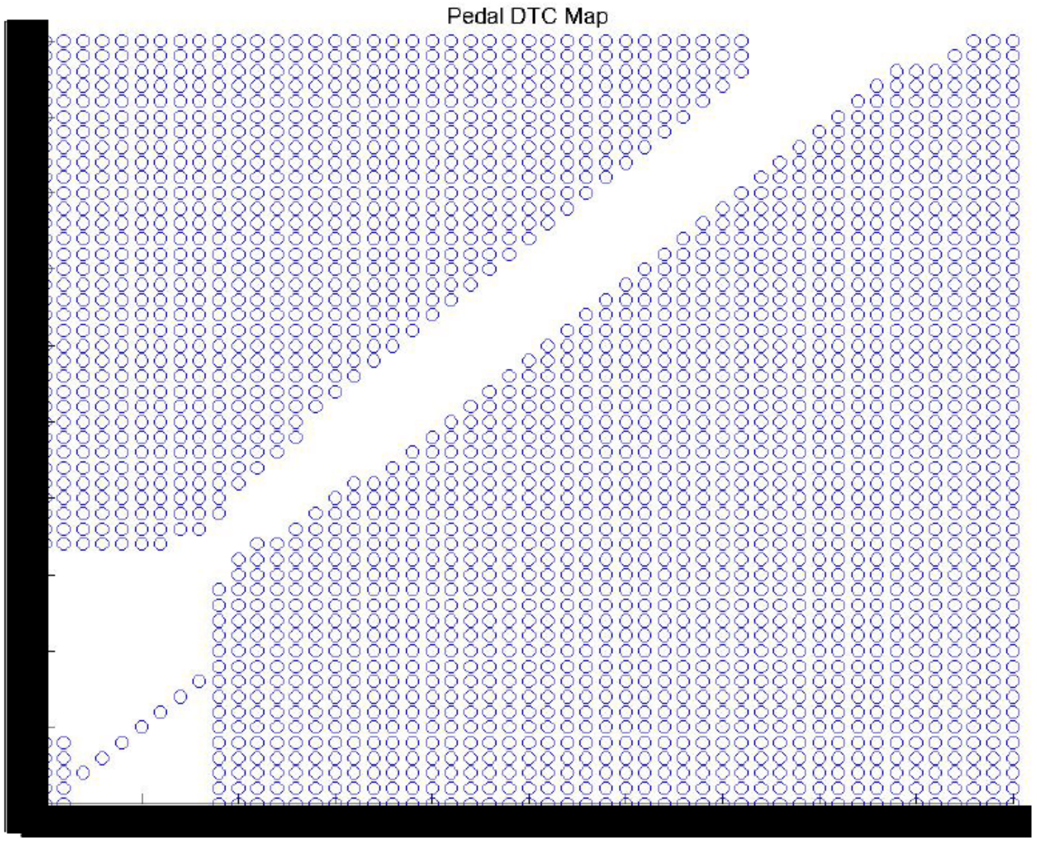


Figure A.11-6. Pedal Diagnostic Map for 2 second Ramp Sequence (Nominal)

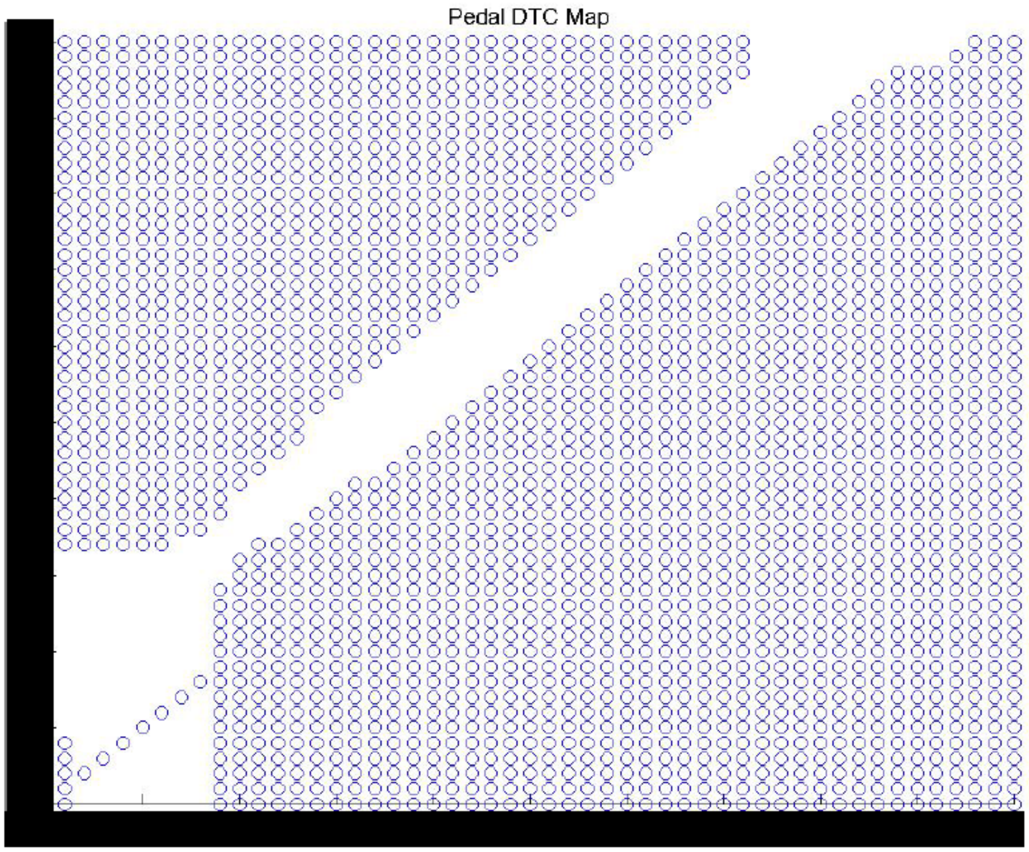


Figure A.11-7. Pedal Diagnostic Map for 2 second Ramp Sequence(Battery Voltage Applied)

The step sequence, similar to the ramp sequence, starts and holds VPA1 and VPA2 at their nominal voltages for 3 seconds, and then, instantaneously steps to a new VPA1 and VPA2 value, which it holds for 2 seconds. This software test produced two maps similar to Figures A.11-6 and A.11-7.

The constant sequence, unlike the ramp and step sequences, starts and holds VPA1 and VPA2 at a set of voltages for 7 seconds. This software test produced two more maps similar to Figures A.11-6 and A.11-7.

A.11.1.4 Sequence to Increase Throttle

By combining the information from the pedal learning and diagnostics, the software team developed a map of all the points of VPA1 and VPA2 starting values, which lead to a learned the minimum fully released pedal value of 10 degrees and 30 degrees for VPA1 and VPA2, respectively. The combinations of values leading to a VPA1 of 10 degrees are shown below in Figure A.11-8.

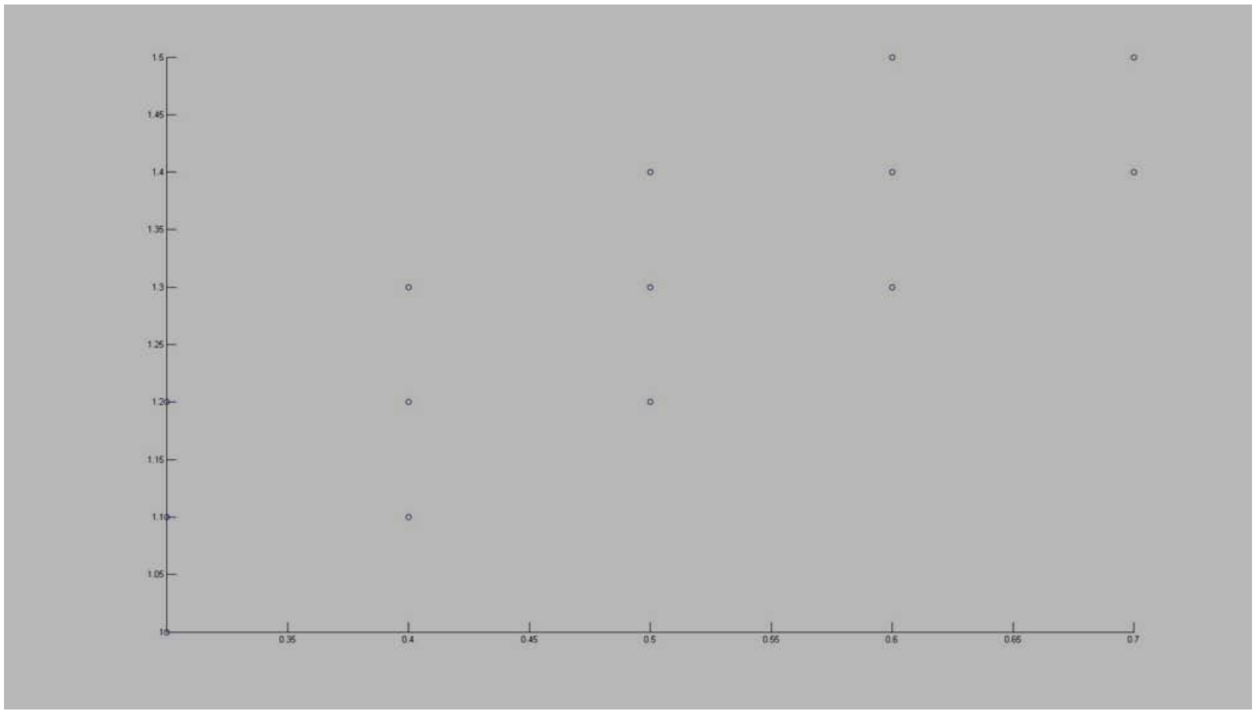


Figure A.11-8. VPA1 and VPA2 Combinations Leading to a Minimal Learned Value after Starting at Nominal

These software model tests and the maps generated became the basis for much of the pedal tests done on the vehicle hardware.

A.11.2 Throttle Control

The throttle is controlled by the ETCS-i software in the form of four signals (HI, HO, LI, LO). These four signals open or close transistors in an H-Bridge IC. The 4 signals are based on conversion from a calculated duty cycle command coming from the PID control software. The duty cycle dictates the closing/opening rate which is controlled by changing the on and off times of four FETs.

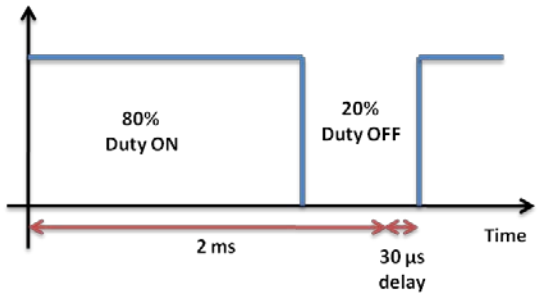


Figure A.11-9. Duty Cycle Times. 80% Duty Example

[redacted]

The input throttle command, which the PID controls to, is a combination of the throttle request from the pedal/cruise/VSC, the request from the Idle Speed Control, and the learned throttle spring position.

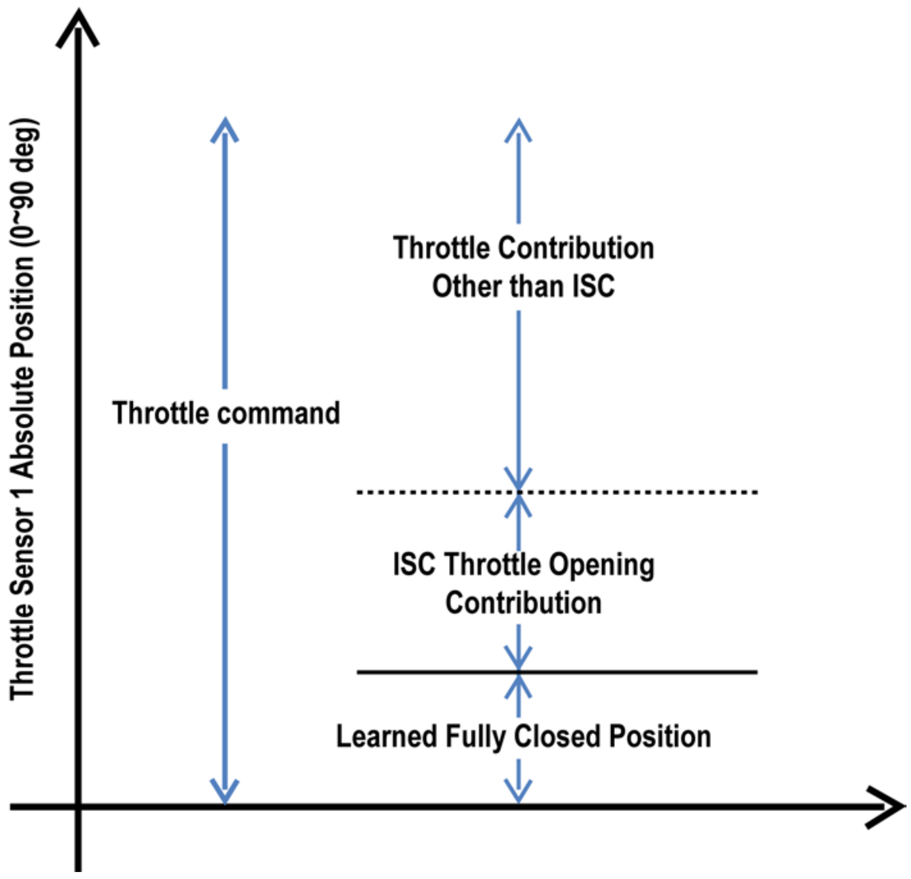


Figure A.11-10. Throttle Command Components

The base for the throttle command comes from the learned spring detent value. This value represents the position of the spring when it is not actively controlled. This spring detent value is not the fully closed position. The fully closed value is "learned" from 20 ms after ignition to 60 ms after ignition, when power is not applied to the throttle valve and it is assumed to be held open by the spring only. [redacted]

[redacted]

[redacted]

[redacted]

The learned fully closed value is the foundation for the determination of all other throttle control, including diagnostics. Like the pedal, the learned throttle value is used in the determination of thresholds. However, unlike the pedal, if the throttle diagnostics determines the existence of a failure, the learning is not reset.

Based on individual throttle sensor and sensor to sensor correlation, checks are performed to determine the validity of the sensor data entering the CPU. To effectively understand and evaluate the range/area of valid or invalid values, the software team used the model to generate plots or "maps" of the two throttle position sensors. The horizontal axis of these maps is VTA1 voltage, and the vertical axis is the VTA2 voltage. A sweep through all possible voltage relationships was performed. Where a software diagnostic detected an invalid condition, it was noted on the map.

From this map, the valid acceptable VTA1 and VTA2 voltages were identified, and all regions where the software would detect invalid inputs were identified.

Three maps were generated from the Matlab model for both the pedal diagnostics and pedal fail-safe detection. Each map presented a difference sequence to the diagnostic logic, including a ramp, step, and constant.

[redacted]

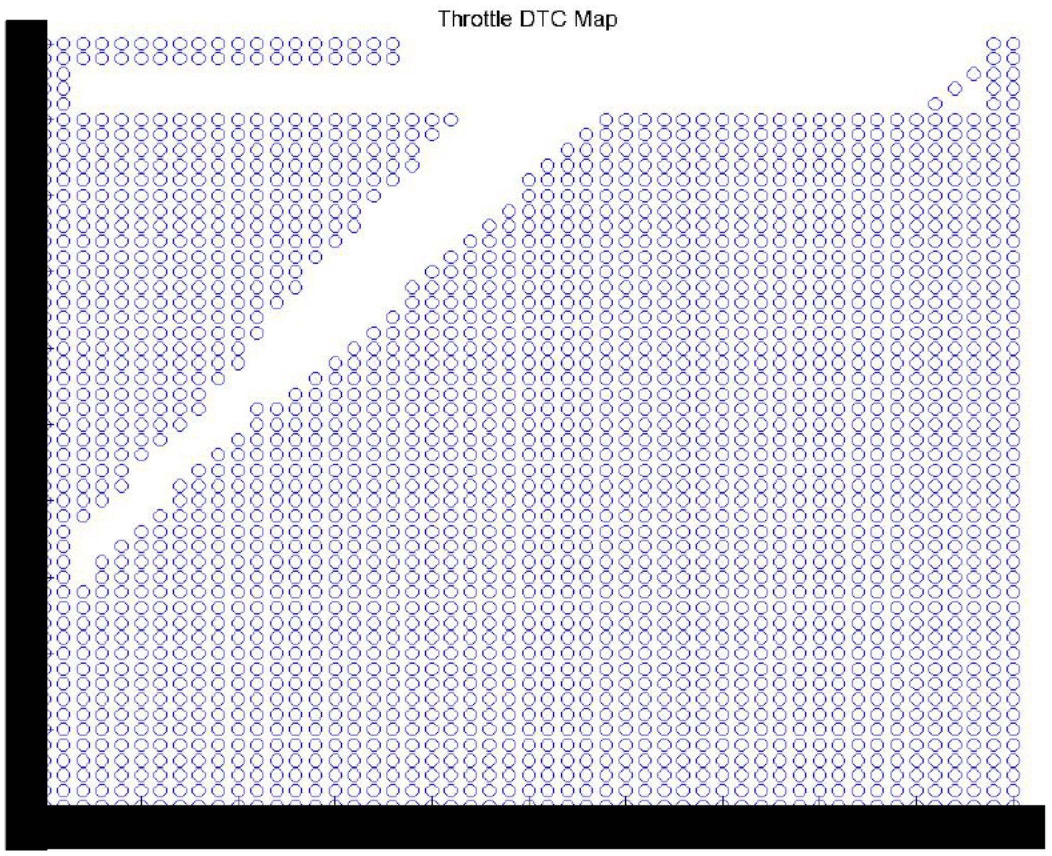


Figure A.11-12. Throttle Diagnostic Ramp Sequence Software Test

The step sequence, similar to the ramp sequence, starts and holds VTA1 and VTA2 at their nominal voltages for 3 seconds, and then, instantaneously steps to a new VTA1 and VTA2 value, which it holds for 2 seconds.

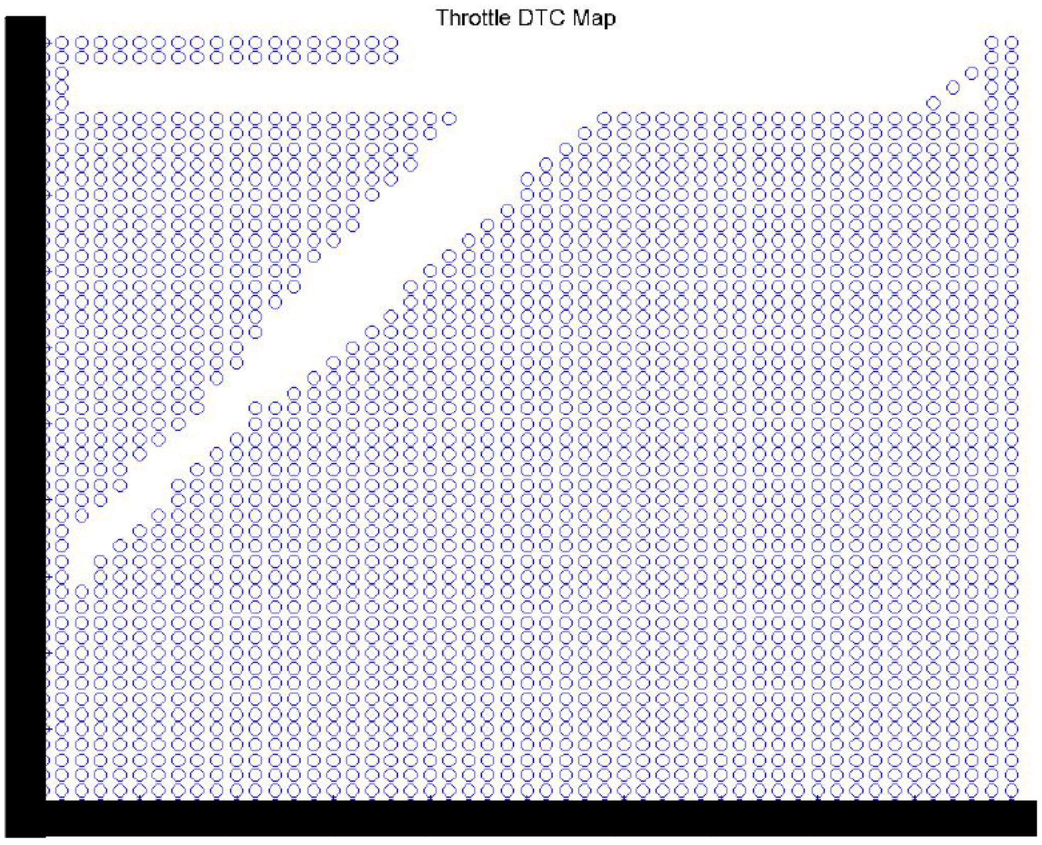


Figure A.11-13. Throttle Diagnostic Step Sequence Software Test

The constant sequence, unlike the ramp and step sequences, starts and holds VTA1 and VTA2 at a set of voltages for 7 seconds.

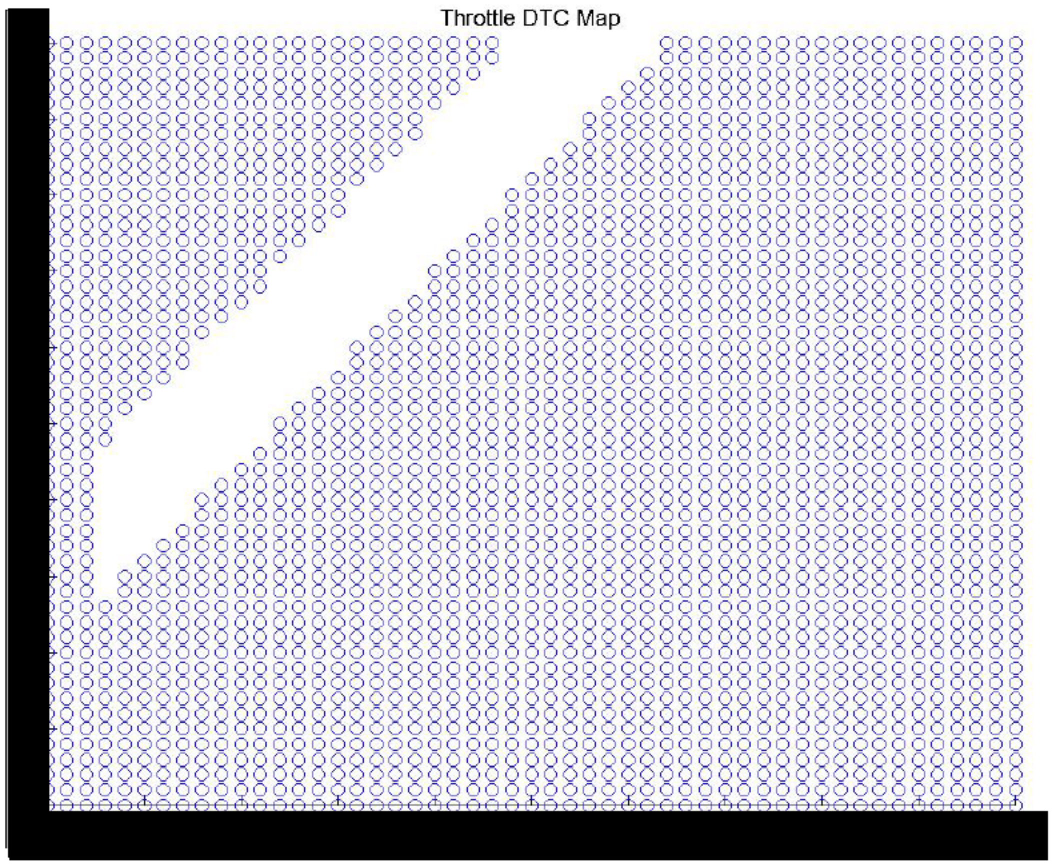


Figure A.11-14. Throttle Diagnostic Start and Hold Software Test

It should be noted that it required these three differing software tests to capture the throttle diagnostic behavior. It should also be noted, when the throttle diagnostic logic determines the existence of a sensor fault, the power to the throttle motor is removed and the throttle will return to a spring detent position.

A.11.3 Electronic Throttle Valve Model

An "Electronic Throttle Valve Model" was provided by TMC. The model consists of a DC motor model, throttle gear/valve model, and a sensor model. The PID software model was derived from documentation and a critical review of the actual vehicle source code.

The main function of the throttle PID controller is to drive and hold the throttle valve in the demanded position. The actual throttle position is returned to the PID controller, and the difference between the actual and the demanded position produces the drive. Small differences are summed in the integral portion of the controller to produce enough drive for the correction. If this summation of errors continues to increase (for example due to a mechanical failure in the throttle) the drive current will continue to increase. If the drive current crosses the hardware sensed maximum, the motor current and drive is removed from the throttle motor, and the throttle springs return the throttle to the closed position.

A.11.3.1 DC Motor

The DC Motor was modeled as a simple LR circuit with deadband and counter electromotive force. In addition to the simple motor model, the DC motor model includes models of spring force, cogging, torque hysteresis, detent, viscous damping, and inertia. The DC Motor model produces the angular rotation rate of the motor shaft.

A.11.3.2 Throttle Gear/Valve

The Throttle Gear and Valve model integrates the motor shaft rotation rate to produce motor shaft angle. The motor shaft angle is multiplied by the gear reduction ratio to compute the resultant throttle valve angle.

A.11.3.3 Throttle Sensor

The throttle sensor is modeled as first order system with transport delay and offset bias.

A.11.3.4 Controller Description

The controller has been documented as a PID [redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

A.11.4 Cruise Control

A.11.4.1 Cruise Control Switch

The cruise control is implemented through a single voltage input that is manipulated by the driver through a switch. Depending on the input from the driver, the switch will select a resistance that is interpreted by the cruise control logic to set the state of the cruise control. Four resistors in parallel that may be combined together to represent the five cruise control switch states: Main, Resume, Set, Cancel, Off. Input from the driver is sensed and considered valid if three consecutive input samples indicate the same state change. The table below describes the conditions for setting of each of the cruise control switch states.

Table A.11-1. Cruise Control Switch Voltage

Cruise Control Switch Voltage	Cruise Control Switch State
CC voltage (CCV) <= (0.168 * Battery Voltage(BV))	Main On/Off
(0.168 * BV) > CCV >= (0.3685 * BV)	Resume On/Off
(0.3685 * BV) > CCV >= (0.584 * BV)	Set On/Off
(0.584 * BV) > CCV >= (0.7934 * BV)	Cancel On/Off
(0.7934 * BV) > CCV	Off

There are noise removal functions to smooth out signal irregularities. Note there are no dead-band voltage ranges between Cruise Control Switch States.

The actual setting of a cruise control state requires one of the cruise control switch states to be registered followed by the "Off" state. When the driver engages the cruise control switch, the switch is pushed in – this corresponds with Main, Resume, Set, or Cancel. When the driver lets off of the switch, it returns to the normal position - this corresponds to the "Off" state.

In addition to the cruise control states described above, there are time-sequence manipulations of the same cruise control switch that allow for other states that are described in the table below.

Table A.11-2. Cruise Control Activation

Cruise Control State	Activation	Description
Coast	Set switch is engaged for longer than 0.6 seconds	While engaged, coast will decrease the speed of the vehicle. When disengaged the new vehicle speed becomes the set speed.
Tap Down	Set switch is engaged	Each time the set switch is engaged the vehicle speed will decrease by 1.6 kph. If the new vehicle speed is more than 5 kph different than the set speed at disengagement, then it becomes the new set speed.
Accel	Resume switch is engaged for longer than 0.6 seconds	While engaged, accel will increase the speed of the vehicle. When disengaged the new vehicle speed becomes the set speed.
Tap Up	Resume switch is engaged	Each time the resume switch is engaged the vehicle speed increases by 1.6 kph. The new vehicle speed does not get saved as the set speed.

The cruise control operation may be manually canceled through four different inputs:

1. Brake is depressed
2. Shift from Drive
3. Cancel switch is engaged
4. Main is turned off

A.11.4.2 Cruise Control Diagnostics

There are four diagnostic codes that describe the cruise control failures.

Table A.11-3. Cruise Control Diagnostics

P0571 Brake Switch Circuit Abnormal	Checks coherency of the two brake switches.
P0500 Vehicle Speed Sensor Abnormal	Checks whether a speed pulse is registered by the vehicle within 140 seconds of ignition on.
P0503 Vehicle Speed Sensor Intermittent/Erratic/High	Checks whether vehicle speed reading changes more than 25% from one reading to the next.
P0607 Cancellation Circuit Abnormal	Checks various voltages, data mirrored in RAM, and brake switch state. Voltages checked include +B low voltage, ignition switch low voltage, WI low voltage, and STA low voltage.

A.11.4.3 Auto Cancel Functions

Auto cancel refers to the function of automatically canceling the cruise control set speed because of certain conditions or diagnostic output. There are three subsets of auto cancel described in the Table A.11-4.

Table A.11-4. Cruise Control Auto Cancel

C1 Low Speed	Cancels when the vehicle speed is less than 36 kph, or 16 kph below the set speed.
C2 Diagnostics(No code)	Cancels when there is an abnormality detected in the electronic throttle or there is a contradiction in the two accelerator pedal position sensors or if there is an abnormality in the accelerator pedal position sensors, or there is an abnormality in the intake air mass flow valve or if the data mirrored in RAM is not nominal.
C3 Diagnostics(P0571, P0500, P0503, P0607)	Cancels if any of the following diagnostic codes occur: (P0571, P0500, P0503, P0607).

A.11.5 Idle Speed Control

The Idle Speed Control (ISC) is a feed forward control system, working to mainly set the value for idle rpm. In addition the ISC controls functions to compensate for conditions like creep control, increases in oil temperature, variable valve timing, alternator loads, air conditioner loads, catalyst temperature, idle while moving, stall prevention, fuel cut, variations in the throttle valve assembly, purging, power steering, startup/ignition, and engine temperature to smooth the driving experience engine operation. The ISC throttle angle request is added to the throttle requests from the other control functions. ISC calculates in terms of the amount of air required, in Liters/second, and converts this value to a throttle angle request. Within the ISC function there is a conversion mapping, in the form of a look-up table, which converts the amount of air requested to a throttle angle. The final throttle angle request includes a learning value to compensate for deposits in the throttle assembly. The ISC contribution is comprised of three main components: 1) The ISC learning compensation, 2) the ISC target rpm/actual rpm feedback control, and 3) engine loads. The table below describes calculated maximum contributions from each of the ISC modules.

[redacted]

There is a maximum ISC throttle angle contribution of 15.5 degrees. Testing of the model (258,048 iterations), showed a maximum contribution of 11.557 degrees assuming a base value concerning throttle valve deposits and manufacturing variances. If deposits accumulate over time, the contribution may reach the maximum guard of 15.5 degrees.

A.11.5.1 ISC water temperature measurement usage and fuel cut

The ISC uses water temperature measurement, as an input to various software modules within the ISC to determine throttle angle contribution to maintain idle. Many of these calculations are done based on the measured water temperature. There are no redundant measurements referenced within these modules to provide verification of the water temperature. If the

measurement is compromised, an increase in throttle angle can be demanded from ISC to the throttle motor.
These findings from the software models influenced the development of tests performed on vehicle hardware.

A.11.5.2 "Idle On" Fuel Cut Function

Whenever the pedal is sensed to be released, the engine is commanded to idle. This occurs when the vehicle is stopped or in motion.

At the moment the pedal is released, the rpm is sensed. If this rpm is above the fuel cut threshold, fuel is cut from the engine, and the engine rpm begins to decrease. When the rpm decreases below the fuel cut return threshold, the fuel is returned to the engine to maintain proper idle speed.

The fuel cut function will engage when the pedal is released and the rpm is above the fuel cut threshold. [redacted]

[redacted]

	[redacted]			[redacted]			[redacted]			[redacted]	
	[redacted]			[redacted]			[redacted]			[redacted]	
	[redacted]			[redacted]			[redacted]			[redacted]	

[redacted]

Once fuel cut is engaged, the rpm will drop until it reaches a fuel cut disengage limit. [redacted]

[redacted]

	[redacted]			[redacted]			[redacted]			[redacted]	
	[redacted]			[redacted]			[redacted]			[redacted]	

[redacted]

[redacted]

[redacted]

A.11.5.3 ISC Learning Algorithm Description

The Idle Speed Control (ISC) learning algorithm is designed to maintain smooth idling conditions. [redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

A.12 Model Verification

Findings and recommendations were limited to test results involving Toyota hardware. Any observations or findings that were a result of Matlab model analysis had to be verified through hardware testing. Since the software analysis was used to aid in hardware testing scenario development, verification of Matlab models was advantageous. The first verification of model behavior was by the Toyota and Denso subsystem experts. Next the model was exercised through automatic software testing. If the model testing revealed unexpected behavior, further investigated was warranted. In some cases, the unexpected behavior was either found to be a modeling issue or was passed on for hardware testing. The ultimate verification of the model behavior was the Camry hardware testing corroboration.

All pedal maps, pedal learning behavior, throttle maps, and throttle learning behavior were tested in the modeling environment, and confirmed using the actual Toyota source code within the Greenhills development environment. The insight gained from these tests provided input and direction to the vehicle testing.

A.12.1 Idle Speed Control (ISC) Model Completeness

The process of verification for the ISC model takes a two-fold approach. First, Matlab models were functionally verified by inspection by a Toyota ISC specialist. Next, model predicted behavior was compared to test article behavior during hardware testing.

The model requires a variety of inputs from the rest of the vehicle. The challenge with the testing of the ISC is that the throttle angle command that is output effects the engine and therefore changes the inputs that that control is based on. This necessitated an engine plant model to correctly represent the behavior of the engine given the throttle angle demand. During the process of this investigation no such engine model was made available. There is a vehicle model that was developed by Toyota to support the testing of the cruise control that provides a limited amount of feedback that was used to test the ISC model. Even the limited amount of feedback that the vehicle model provides is not appropriate for ISC testing for certain conditions. The vehicle model was developed for a vehicle that is in motion and being propelled through an engaged transmission. The ISC behavior is most significantly observed during “idle-on” condition where there is no input to the throttle angle from commands initiated by the cruise control or the foot-pedal. A fraction of “idle-on” condition may occur with the conditions represented by the vehicle model, but the majority of the ISC operation is tested at low speed conditions where the feedback from the partial plant would not be complete and may even provide incorrect feedback that would induce unexpected behavior through the model. This was taken into consideration when testing occurred.

The model inspection process was initiated with the identification of a Toyota software module of interest. The identified software module was then exercised by varying the inputs across a range of values whereupon the output was inspected for correct behavior. The Toyota ISC specialist, having insight into the system, was qualified to decide what the significant inputs to the software module were to verify functionality. The outputs were observed as changes from a baseline behavior that was decided on at the beginning of testing. This baseline behavior represented fairly benign operating conditions for the ISC with the majority of the throttle angle command contribution coming from EQG, which represents the learned value for throttle position. The software modules were inspected individually at first and then combinations of modules were exercised. The combinations were based on the identification of functional interactions between separate software modules.

The testing process described was a good first step in verification of model behavior, but the behavior is only tested for a limited amount of inputs and interactions as compared to a comprehensive set of tests. A complete set of tests would be physically impossible for inspection by engineers, but automatic testing can be implemented to gain more confidence in the model. Matlab System Test was implemented to more fully exercise the model. The tests were implemented by varying inputs that have significant effect on the throttle angle contribution. The relevant combinations that lead to a large throttle angle command from the ISC system were then identified. While the tests themselves do not give evidence for verification of model behavior, any combination of inputs that lead to large throttle commands may be investigated in more detail to verify behavior.

Verification by inspection and automatic testing provides a level of confidence in model behavior for hardware scenario test development, but another source of model verification is completed when the hardware tests of the ISC exhibit the same behavior predicted in the model. All ISC functions outside of the learning algorithm have been verified through inspection and minimally tested through SystemTest. Future verification work could include a more comprehensive set of automatic software tests and hardware tests with the vehicle.

A.12.2 ISC Learning Algorithm Verification

Complete verification of the ISC learning algorithm functionality did not reach maturity. While the verification exercises outline above provided a good measure of confidence in most of the software modules of the ISC, EQG(learned value), EQI(feedback value) and EQGRST(learning reset) provide unique challenges for verification. The learning algorithms represented by these three modules are incredibly complex, requiring many inputs from inside and outside the ISC being utilized by multiple, interconnected calculations.

A.13 Software Model Testing

The objective of these tests was to use the MATLAB, SIMULINK and StateFlow models that have been developed from the Toyota software specifications to find scenarios that could lead to UA. If found, the scenarios would subsequently be tested on the Toyota source code within the Green Hills environment, as well as on the actual hardware to see if they could lead to an actual UA.

There were three main sets of software tests conducted:

1) varying pedal sensor input voltages, VPA1 and VPA2, to determine the effects of transient voltage changes and the resulting effect on the throttle.
2) determining what conditions may cause cruise control to turn on unexpectedly as well as what conditions prevent the cancelling of cruise control.
3) determining the maximum ISC contribution.

The SystemTest tool from Mathworks was used to conduct these model tests. The tool encapsulates the model, its initial setup, test vectors used and the properties that are checked. The tool runs the model with thousands or even hundreds of thousands of test cases using a permutation of all the input test vectors. The tool also generates a complete report of the test data used and the results obtained. The results were then analyzed to come up with one or more UA scenarios for testing on the MY 2005 Camry.

A.13.1 Tests for Failures Resulting from Transient VPA Changes

Test Objective

The objective for this set of tests is to explore the possibility of throttle angle unexpectedly becoming and staying large (e.g., >20) at the end of a trip as a result of a sudden and transient change in the values of VPA 1 and 2 with no diagnostics being detected.

Test Setup

Figure A.13-1 shows the SIMULINK model that was used to conduct these tests. It extends the integrated Toyota system model, Integrated Model (Version 6), by adding:

- The required in-ports to supply the test parameters,
- Blocks used to control the sequence and timing of the input values, and
- The required out-ports needed to channel the output of the application.

The additions are shown in light blue. Another modification that was done to the original model was to remove any SIMULINK display blocks (e.g., Scopes and Displays) to speed up the running of the test.

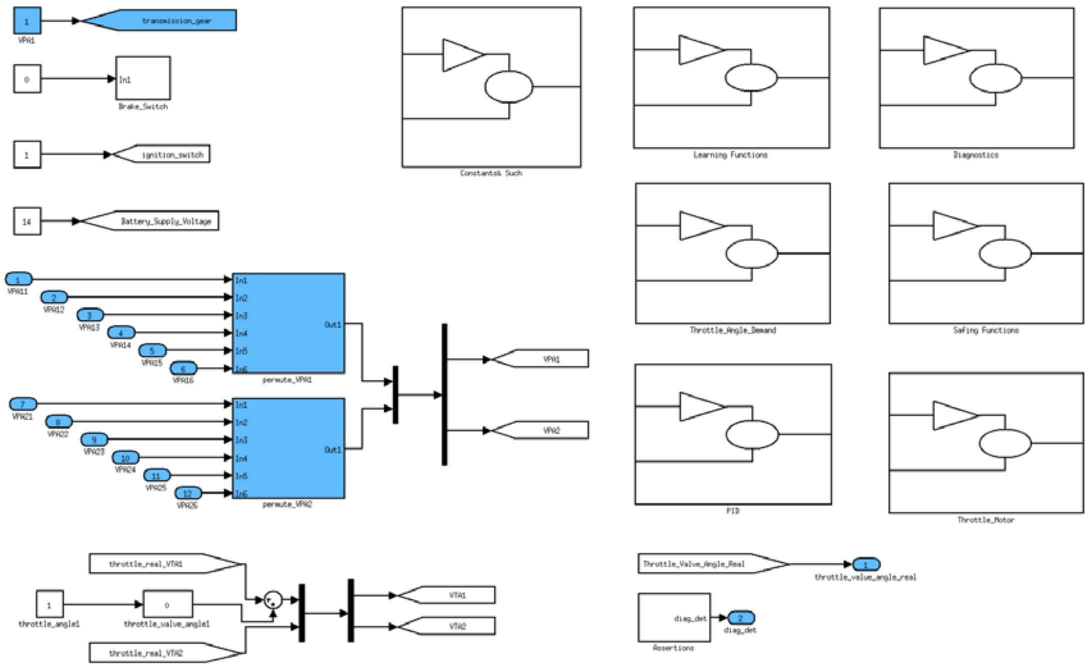


Figure A.13-1. Integrated test model (Integrated_Model_V9_ST.mdl)

The model has a simulation time period of 8 seconds with a time-step of 0.0001.

Trip Plan

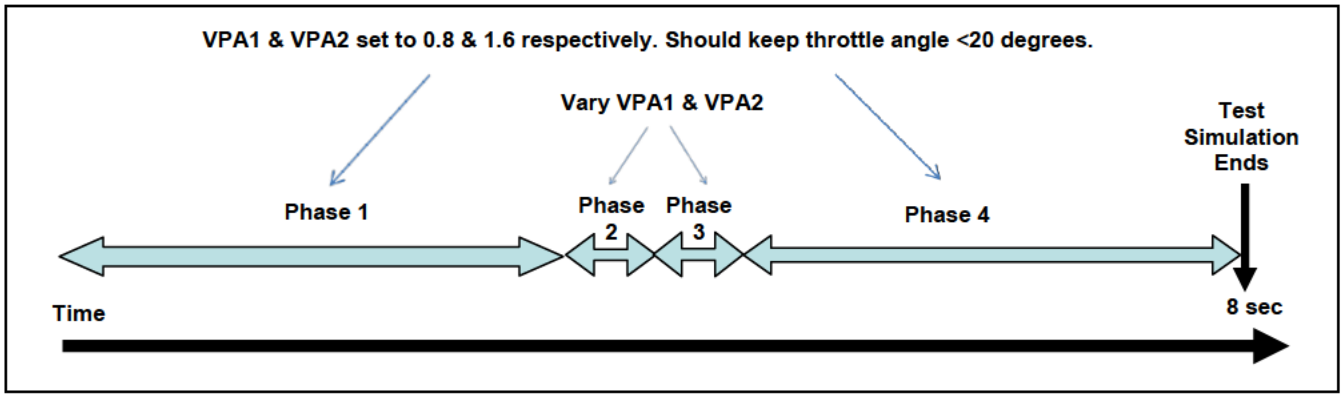

VPA1 & VPA2 set to 0.8 & 1.6 respectively. Should keep throttle angle <20 degrees.
Vary VPA1 & VPA2
Test Simulation Ends
Phase 1
Phase 2
Phase 3
Phase 4
Time
8 sec

Figure A.13-2. Trip Plan

The trip plan simulated by the test model consists of 4 main phases shown in Figure A.13-2.

- Phase 1: t<=4.01
- Phase 2: 4.01<t<=4.02
- Phase 3: 4.02<t<=4.03
- Phase 4: 4.03<t

During Phases 1 and 4, VPA1 and VPA2 are set to normal values, 0.8V and 1.6V, respectively. In this way the trip starts and ends in a normal state and the throttle angle value should be less than 20 degrees. VPA1 and VPA2 are changed during Phases 2 and 3 as shown in the table below. The throttle angle value at the end of the trip is checked.

Test Property

A test fails when the throttle angle value at the end of the trip remain large (>20) and there are no diagnostics detected.

Test Vectors

The vector table shows the VPA1 and VPA2 parameters set from SystemTest using the corresponding test vectors. In Phase 2 and Phase 3, the VPA1 and VPA2 are varied to produce all possible voltage combinations in 0.2V steps.

Input Name	Phase 1	Phase 2	Phase 3	Phase 4
	t<=4.01	4.01<t<=4.02	4.02<t<=4.03	t>4.03
VPA1	0.8	0 to 2.4 in steps of 0.2	0 to 2.4 in steps of 0.2	0.8
VPA2	1.6	0 to 5 in steps of 0.2	0 to 5 in steps of 0.2	1.6

A SystemTest test file was created that references and initializes the test model, sets up the required test vectors and includes the property that is checked during the test. SystemTest runs the tests which in this case are 114,244 test cases representing all permutations of the parameters during trip Phases 2 and 3.

Test Results

A total of 114,244 test cases were run using a cluster of 32 processors that took 13 hours to complete. All the tests passed, i.e., for inputs expected to demand throttle angles less than 20 degrees, none resulted in an increase in the throttle angle value greater than 20 degrees as a result of the transient VPA changes with no diagnostics being detected. The maximum throttle increase observed in the model was around 17.2 degrees.

Note that this set of tests did not explore the values of VPA1 and VPA2 out of the normal ranges (as shown in the vector table).

Further tests

In addition to the tests described above the software team also ran 10,000 tests that explored a wider range of values for VPA1 and VPA2. The software team used random values for these parameters (using a uniform distribution) ranging from -5 to 10. They also all succeeded and did not show any failures.

A.13.2 Testing for Unexpected Cruise Control (CC) Turn-on Failures

Test Objective

The objective for this set of tests is to detect if the Cruise Control system can unexpectedly turn on. In these tests the aim is to generate the set of scenarios in which the user turns on the CC, sets CC speed and then turns CC off at the start of a trip but later on due to some combination of various parameter changes it turns on again unexpectedly.

Test Model

Figure A.13-3 shows the SIMULINK model that was used to conduct these tests. It extends the Cruise Control block/sub-system of the original Integrated Model (Version 6) with:

- The required in-ports to supply the test parameters
- Blocks used to control the sequence and timing of the input values, and
- The required out-ports needed to channel the output of the application.

The additions are shown in light blue. Another modification that was done to the original model was to remove any SIMULINK display blocks (e.g., Scopes and Displays) to speed up the running of the test.

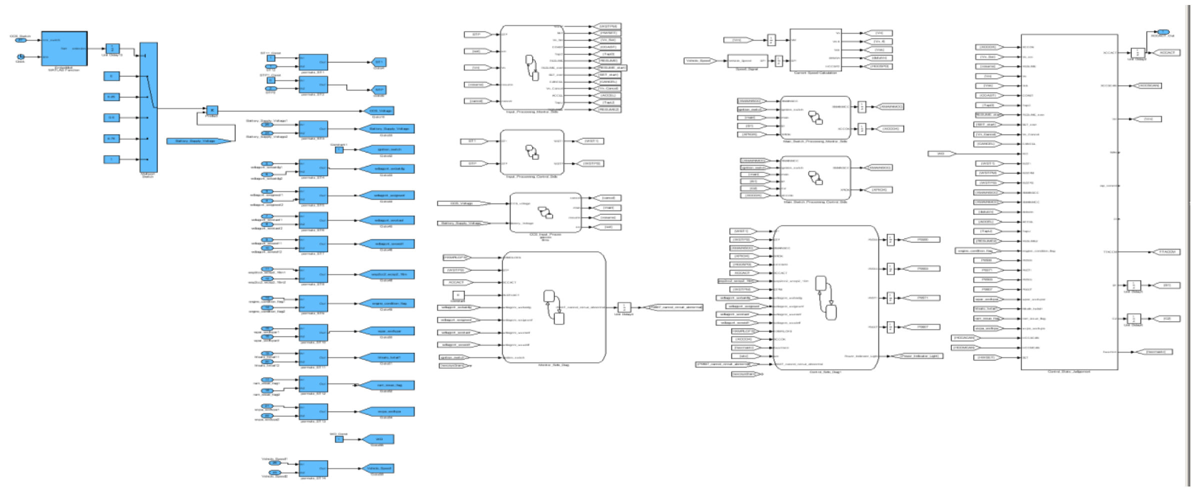


Figure A.13-3. Cruise Control Test Model (CruiseControl_V6_ST.mdl)

The model has a simulation time period of 5 seconds with a time-step of 0.0001.

Trip Plan

The trip plan simulated by the test model consists of 2 main phases, as shown in Figure A.13-4:

- 0<=t<=2: The initial phase in which Cruise Control is turned on, the CC speed is set and then the CC is turned off.
- 2<t<=5: The second phase in which parameters are changed to see if they could violate the property.

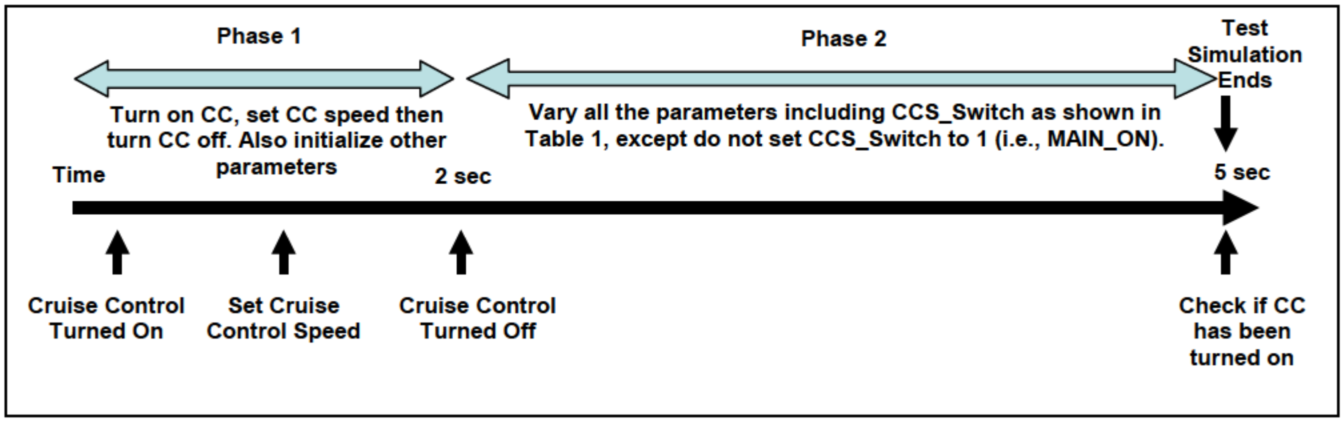

Phase 1
Phase 2
Test Simulation Ends
Turn on CC, set CC speed then turn CC off. Also initialize other parameters
Vary all the parameters including CCS_Switch as shown in Table 1, except do not set CCS_Switch to 1 (i.e., MAIN_ON).
Time
2 sec
5 sec
Cruise Control Turned On
Set Cruise Control Speed
Cruise Control Turned Off
Check if CC has been turned on

Figure A.13-4. Trip Plan

Test Property

A test fails when the last CC output value at the end of the simulation is one. It indicates that CC has turned on unexpectedly.

Test Vectors

The following table shows the parameters that are either set directly in the model or supplied indirectly by the team's automated testing tool (SystemTest).

Table A.13-1. Trip Phases

Input Name	Trip Phase 1						Trip Phase 2
	0<t<=0.25	0.25<t<=0.5	0.5<t<0.75	0.75<t<=1	1<t<=1.25	1.25<t<=2	2<t<=5
Cruise Control Switch Input	Cruise Control On		Set CC speed		Cruise Control Off		
	MAIN ON	MAIN OFF	SET ON	SET OFF	MAI N ON	MAIN OFF	2,3,4,5
	1	5	3	5	1	5	
Brake Switch 1	1						1
Brake Switch 2	0						0
Battery Supply Voltage	14						1,14

Input Name	Trip Phase 1						Trip Phase 2
	0<t<=0.25	0.25<t<=0.5	0.5<t<0.75	0.75<t<=1	1<t<=1.25	1.25<t<=2	2<t<=5
Vehicle Speed	37						20,40
Ignition Switch	1						1
Engine Condition Flag[23]	0						0,1
Flag for system down due to abnormality in Electronic Throttle System	0						0,1
Ram mirror image consistency flag	0						0,1
+B low voltage flag	0						0,1
Ignition Switch Low Voltage Flag	0						0,1
STA Low Voltage Flag	0						0,1
WI Low Voltage Flag	0						0,1
Accelerator Pedal Position Sensor (VPA) 1 and 2 consistency flag	0						0,1

[23] This is a flag that represents abnormality in the intake air mass flow valve. This requires four conditions to be true:
1) XCCACT = 1
2) XCCSCAN = 0 (not under cancelation processing)
3) degree of opening of the intake air mass flow valve is less than 30 degrees
4) intake air mass flow valve sensor is normal, this checks whether the valve is less than 30 degrees or not.

Input Name	Trip Phase 1						Trip Phase 2
	0<t<=0.25	0.25<t<=0.5	0.5<t<0.75	0.75<t<=1	1<t<=1.25	1.25<t<=2	2<t<=5
16 ms Counter from Ignition to Speed Impulse Reception	0						0,1
Flag for Detection of short/intermittent accelerator pedal position sensor	0						0,1
In Drive state flag	1						1

The software team created a SystemTest test file that references and initializes the test model, sets up the required test vectors and includes the property that is checked during the test. SystemTest runs the tests which in this case are 16384 cases which are all the permutations of the values of the parameters during different phases of simulation.

Test Results

The software team ran 16384 test cases consisting of all the permutations of the input parameters during different phases of the simulation shown in Table A.13-1. The test took about 1 hour to complete.

All the tests succeeded, i.e., there were no failures of the property.

A.13.3 Testing for Auto-Cancel Failures

The objective for this set of tests is to detect if the cruise control system may fail to turn off as a result of an auto-cancel condition (e.g., braking). In these tests the aim is to find the set of scenarios in which the driver turns on cruise control and sets the cruise control speed at the start of a trip, but then either he/she performs some actions (e.g., like braking) or the state of the vehicle changes (e.g., some significant diagnostic flags are set) in such a way that should cause CC to automatically get cancelled, but fails to do so.

Test Model

The software team used a slightly modified version of the model shown in Figure A.13-5 to conduct these tests. The previous model was also modified by changing some of the conditions at the start of the trip.

Trip Plan

The trip plan simulated by the test model consists of 2 main phases:

- 0<=t<=2: The initial phase in which Cruise Control is turned on and the CC speed is set.
- 2<t<=5: The second phase in which parameters are changed to see if they could violate the property.

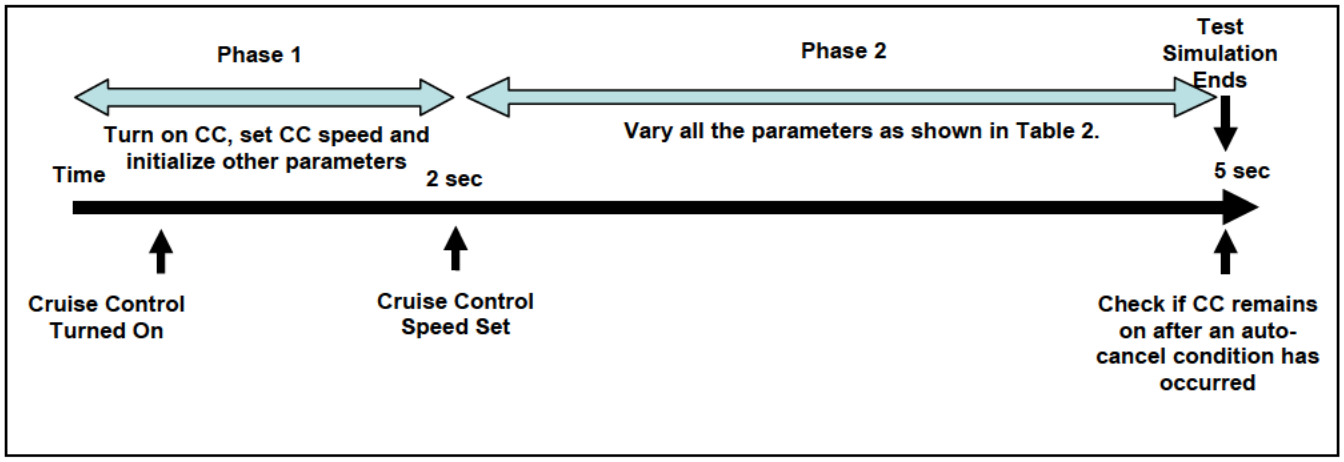

Phase 1
Phase 2
Test Simulation Ends
Turn on CC, set CC speed and initialize other parameters
Vary all the parameters as shown in Table 2.
Time
2 sec
5 sec
Cruise Control Turned On
Cruise Control Speed Set
Check if CC remains on after an auto-cancel condition has occurred

Figure A.13-5. Testing for Auto-Cancel Failure

Test Property

A test fails when the last CC output value (called XCCACT) at the end of the trip is one and any of the following conditions holds in the second phase of the trip:

"Vehicle Speed"<=33

"Vehicle Speed">=65

"Ram mirror image consistency flag"==*1*

"Brake Switch 2"==*1*

"Brake Switch 1"==*0*

A zero output under this condition indicates that CC has turned off as expected and the auto canceling feature worked.

Note that there may be other auto-cancelling conditions that the software team has not covered in this study and they need to be checked in further tests.

Test Vectors

The following table shows the parameters that are either set directly in the model or supplied indirectly by SystemTest.

Input Name	Trip Phase 1 (0<t<=2)	Trip Phase 2 (2<t<=5)
Brake Switch 1	1	0,1
Brake Switch 2	0	0,1
Battery Supply Voltage	14	1,14
Vehicle Speed	40	33,40,65
Ignition Switch	1	1
Engine Condition Flag	0	0,1
Flag for system down due to abnormality in Electronic Throttle System	0	0,1
Ram mirror image consistency flag	0	0,1
+B low voltage flag	0	0,1
Ignition Switch Low Voltage Flag	0	0,1
STA Low Voltage Flag	0	0,1
WI Low Voltage Flag	0	0,1
Accelerator Pedal Position Sensor (VPA) 1 and 2 consistency flag	0	0,1
16 ms Counter from Ignition to Speed Impulse Reception	0	0,1

Input Name	Trip Phase 1 (0<t<=2)	Trip Phase 2 (2<t<=5)
Flag for Detection of short/intermittent accelerator pedal position sensor	0	0,1
In Drive state flag	1	1

As before the software team created a SystemTest test file that references and initializes the test model, sets up the required test vectors and includes the property that is checked during the test (Figure TBD). SystemTest runs the tests which in this case are 24576 cases which are all the permutations of the values of the parameters during the second phase of simulation.

Test Results

The software team ran 24576 test cases consisting of all the permutations of the input parameters during the second phase of the simulation shown in table 2. The tests were run on a cluster of 32 processors and it took about 1 hour and 45 min. to complete. No failures were detected.

Analysis of the Test Results

The tests described above do not cover all the auto-cancel conditions for the CCS, including auto-cancelation related to some of the diagnostics (P0607 – Cancellation Circuit Abnormal). The CCS model did not model and support other off-nominal cases that were outside of the scope of the modeling effort.

A.13.4 ISC Maximum Output Test

Test Objective

The objective of this set of tests is to find the scenarios in which the ISC model output "ettaisc_ettaisc" reaches its maximum value and to determine what that value is. The set of parameters that result in large ISC output would provide good candidates for further investigation.

Test Model

Figure A.13-6 shows the SIMULINK model that was used to conduct these tests. It extends the original ISC model version 3 with:

- The required in-ports to supply the parameters to the test model
- The required out-ports needed to channel the output of the test model

The additions are shown in light blue.

[redacted]

The model has a simulation time period of 4 seconds with a time-step of 0.0001.

Test Vectors

System test was set up to generate different permutations of the parameters shown in the following table.

[redacted]	**Description**	**Range of values**
[redacted]	Catalytic Converter Warmup Control in Crank Angle	[-20,-10,0]
[redacted]	Air temperature in degrees Celsius	[3,20,50,80,100]
[redacted]	Water temperature in degrees Celsius	[-40,-20,-5,0,10,20,30,70,85]
[redacted]	Spark Timing Feedback Control	[0,1]

[redacted]	Description	Range of values
[redacted]	Ambient Air Pressure in Kilopascals	[0.7,1]
[redacted]	Air Conditioner ON	[0,1]
[redacted]	Electrical Load Judgment Flag	[0,1]
[redacted]	Cooling Fan ON	[0,1]
[redacted]	Neutral Switch	[0,1]
[redacted]	Variable Valve Timing Fail Flag	[0,1]
[redacted]	Power Steering Pressure Switch	[0,1]
[redacted]	Brake Signal	[0,1]
[redacted]	Idle Up Low Request when A/T Oil Temperature is High	[0,1]
[redacted]	Idle Up High Request when A/T Oil Temperature is High	[0,1]

The software team created a SystemTest test file that references and initializes the test model, sets up the required test vectors and includes the property that is checked during the SystemTest executes.

Test Property

Although there is no strict pass or fail criteria in this case, in order to more easily detect the interesting cases the software team defined the following test property:

"maximum output value (i.e., max_ettaisc_ettaisc) being greater than 10."

Test cases in which this property is satisfied are marked as passed otherwise they are recorded as failures in the auto-generated report. This allows the team to quickly divide the interesting test results into two major categories. Failed cases can then be further analyzed to detect general patterns and groupings of input parameters that lead to the maximum output value.

Test Results

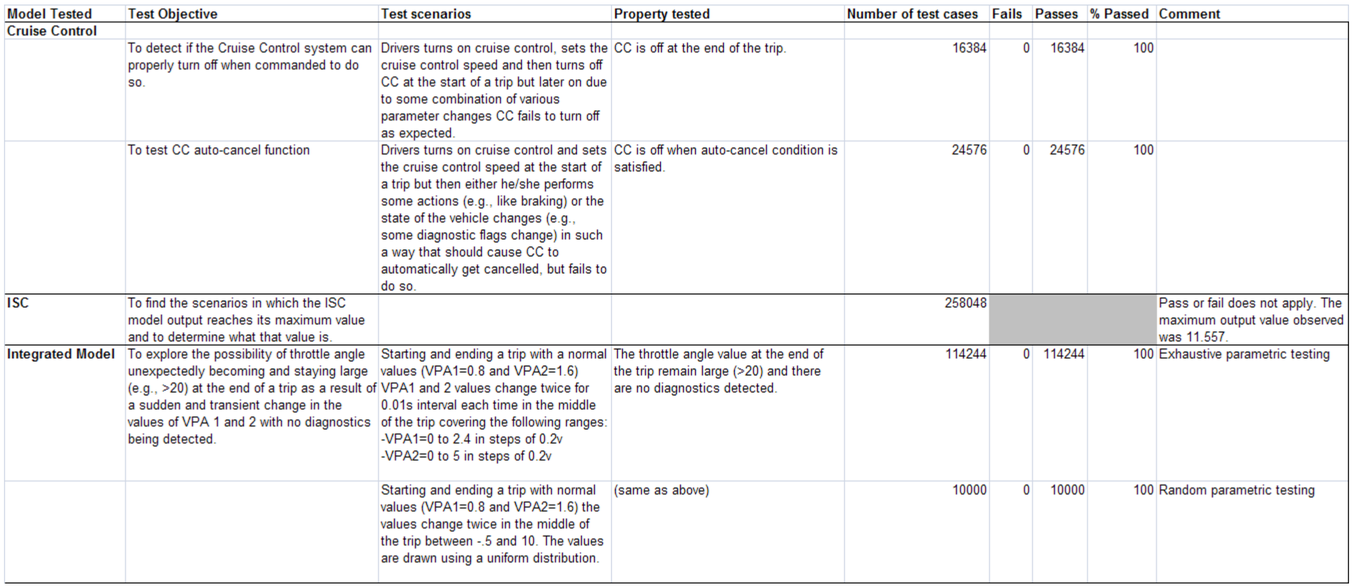
The software team ran 258048 test cases out of the total of 276480 test cases. The tests were run on a cluster of machines with 32 processors and took around 42 hours to complete. By default System Test saves the test data including the values of parameters used to test the model in a MATLAB ".mat" file. In addition it generates an HTML report which shows the test cases and some general information like the start and stop time of the test, summary pass/fail reports, etc.

Analysis of the Results

There were 183840 test cases (around 70%) that showed output values of less than 10 and 74208 cases (around 30%) showed output values of greater than 10. The maximum output value observed in the software model was 11.557. These test cases show that the maximum output value is correlated with the minimum water temperature sensor values. A closer examination of all the results may be necessary to detect any other relationship that might exist between the parameters in other test cases.

2005MY Camry vehicle hardware tests were performed with water temperature sensor failures as a result of this software model test.

A.13.5 Summary of System Test Results



Model Tested	Test Objective	Test scenarios	Property tested	Number of test cases	Fails	Passes	% Passed	Comment
Cruise Control								
	To detect if the Cruise Control system can properly turn off when commanded to do so.	Drivers turns on cruise control, sets the cruise control speed and then turns off CC at the start of a trip but later on due to some combination of various parameter changes CC fails to turn off as expected.	CC is off at the end of the trip.	16384	0	16384	100	
	To test CC auto-cancel function	Drivers turns on cruise control and sets the cruise control speed at the start of a trip but then either he/she performs some actions (e.g., like braking) or the state of the vehicle changes (e.g., some diagnostic flags change) in such a way that should cause CC to automatically get cancelled, but fails to do so.	CC is off when auto-cancel condition is satisfied.	24576	0	24576	100	
ISC	To find the scenarios in which the ISC model output reaches its maximum value and to determine what that value is.			258048				Pass or fail does not apply. The maximum output value observed was 11.557.
Integrated Model	To explore the possibility of throttle angle unexpectedly becoming and staying large (e.g., >20) at the end of a trip as a result of a sudden and transient change in the values of VPA 1 and 2 with no diagnostics being detected.	Starting and ending a trip with a normal values (VPA1=0.8 and VPA2=1.6) VPA1 and 2 values change twice for 0.01s interval each time in the middle of the trip covering the following ranges: -VPA1=0 to 2.4 in steps of 0.2v -VPA2=0 to 5 in steps of 0.2v	The throttle angle value at the end of the trip remain large (>20) and there are no diagnostics detected.	114244	0	114244	100	Exhaustive parametric testing
		Starting and ending a trip with normal values (VPA1=0.8 and VPA2=1.6) the values change twice in the middle of the trip between -.5 and 10. The values are drawn using a uniform distribution.	(same as above)	10000	0	10000	100	Random parametric testing

A.14 Analysis of Real-Time Software Behavior

A.14.1 Introduction

The Toyota ETCS-i is an example of a safety-critical hard real-time system. In a hard real-time system, the hard-deadline timeliness of the software is just as important as its correctness. The following sections present an analysis of the ETCS-i from this perspective.

The ETCS-i is a complex system that consists of many components and thousands of lines of code. This section highlights some key characteristics of the system's hardware and software. Although the details are presented here without discussion, their importance will become clear in subsequent sections.

A.14.2 Target Hardware

Like most embedded systems, the hardware is divided into several chips that work together. There is a chip for collecting analog data signals, and another one for managing power. The main chip, on which the software runs, is based on the Renesas Electronics V850E1, a 32-bit RISC CPU running at 64 MHz. It is a proprietary package (part #µPD70F3152) with a unique memory and peripheral layout that is customized to Toyota's specifications.

A.14.3 Embedded Software

The ETCS-i software is implemented in C with certain performance-critical sections in V850 assembly. There are no floating-point operations, and dynamic memory allocation (*malloc*, *free*) is never used after system initialization.

Task scheduling is handled by an underlying operating system compliant with the OSEK specification. Although OSEK offers real-time capabilities (priority ceiling protocol, rate monotonic scheduling, etc.), there is no explicit concept of task deadlines. Tasks are assigned a priority and run periodically (e.g., every four milliseconds) or in response to an event such as an interrupt.

A.14.4 Toyota's Software Performance Testing

Toyota's approach to software testing consists of unit testing and hardware-in-the-loop field testing. Functions are tested in isolation for correctness, and the integration and performance testing is performed empirically on the actual hardware. All verification of timely behavior is accomplished with CPU load measurements and other measurement-based techniques.

Toyota designed the software with a high margin of safety with respect to deadlines and timeliness. For example, a task with a one-millisecond period is intended to consume a small fraction of that time slice (around 100 microseconds). Toyota documented no formal verification that all tasks actually meet this deadline requirement.

Documented tests have shown that the Main CPU has less than 20% idle time at a rpm of 5000 or higher (see Homework #47).

A.14.5 Statistical Analysis

The first timing analysis performed in this study was based on a statistical model of CPU load. The goal was to determine the probability of the load exceeding the 100% threshold (0% idle time). Should such a condition occur, system-critical tasks would be starved for computational resources, possibly leading to spontaneous system resets.

A.14.6 Toyota's Analysis

Prior to the statistical analysis in this study, Toyota provided three types of data:

- Mean percentage of time used by the idle task under different load conditions corresponding to 6000, 8000, and 9000 rpm
- Minimum and maximum margin for the four-millisecond task sampled for ten seconds
- An observation that no CPU reset had occurred during hardware-in-the loop testing

However, none of this data provided sufficient confidence to extrapolate to the approximately billion hours that the Camry fleet averages each year.[24] Either the sample size was too small (e.g., ten seconds for minimum and maximum), or the summary statistic was too coarse (e.g., average idle task percentage).

A.14.7 Empirical Analysis

This study requested more data in order to extrapolate to probabilities on the order of a billion hours per year. Toyota ran the 2005 ECU configuration, with the diagnostic tool connected, on their MITY test stand in Japan, in the three rpm levels for an hour each. The actual margin for each sample of the four-millisecond task was recorded. This required the MITY facility because of the need to provide 250 samples per second for an hour (900,000 samples per hour), compared to the minimum and maximum statistic for ten seconds provided previously. The three configurations were 6000 rpm, 8000 rpm, and 9000 rpm. While 6000 rpm is the maximum engine rpm under normal circumstances before fuel ignition cutoff is engaged, higher rpm is enabled under laboratory conditions. The higher rpm configurations provide a stress test for crosschecking the 6000 rpm data.

A.14.8 Analysis of 6000 RPM Data

The 6000 rpm data contains 3,616,246 samples of margin times in microseconds. The samples were collected every millisecond, and hence there are repetitions for three, four, and five consecutive samples. The data was filtered to remove these repetitions, and the filtered data contained 880,571 samples. In this dataset, any task may have its lifecycle divided into three components:

Waiting time in queue + Execution time + Margin time = four milliseconds

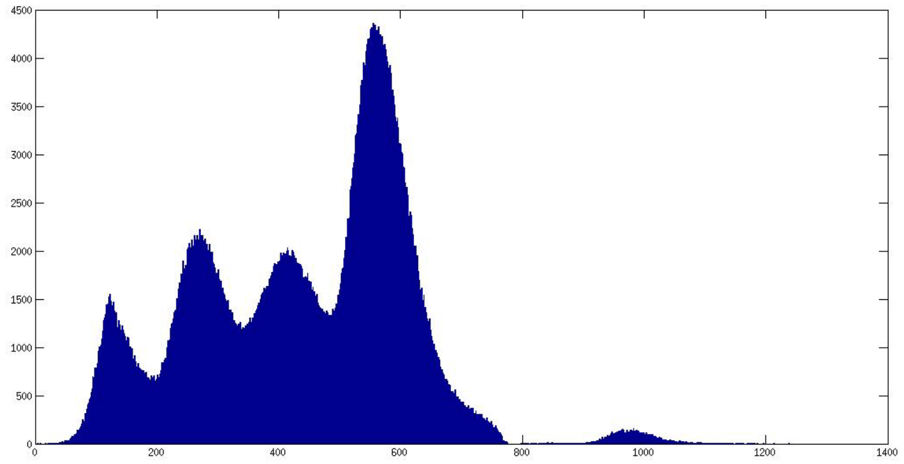
[24] According to the US DOT Federal Highway Administration from 2007 US data, the average annual miles per vehicle is 12,334 per year. This is based on a total annual miles traveled (3,049,047 million) divided by the total number of vehicle registrations (247 million). From 1983 through 2007 over ten million Camrys were sold worldwide. The ECU configuration for 2005 Camry was similar to those from 2002 onwards. From these figures, the rough order of magnitude of hours driven yearly on Camrys from 2002 to 2007 model yearly is one billion hours.

Waiting time refers to the time a task waits in the queue (either due to an interrupt or due to the CPU being used by a higher priority job); *execution time* is the time it takes the job to finish execution; and *margin time* is the time between the completion of one job and the invocation of the next.

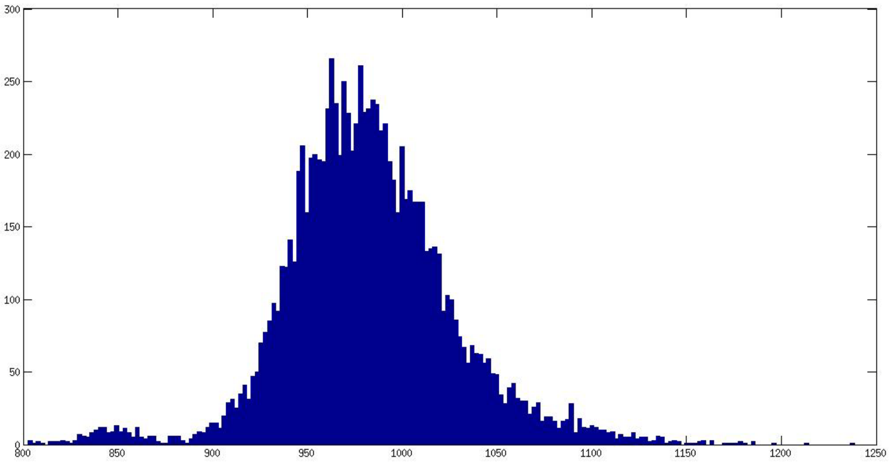
The data supplied by Toyota consisted of the margin times. The execution time was assumed to be the one with the highest margin time (assuming the task finished earliest), subtracted from four milliseconds. The approximate value of execution time is 121 microseconds. The histogram plots below show the waiting times for the four-millisecond tasks, where the horizontal axis corresponds to time and the vertical axis to the frequency. From the plot, it is clear that most of the observations have a waiting time of less than 800 microseconds.

This data is highly multi-modal. The first aim was to fit a mixture distribution to this data. However, a mixture of Gaussians had a bad fit (based on maximum likelihood approach) and so the software team focused solely on the samples that have waiting times greater than 800 microseconds.

The following histograms plot the waiting times for the four-millisecond tasks for 6000 rpm.



The above histogram shows the waiting times (greater than 800 microseconds) of the tasks. This resembles a Gaussian distribution.



The following is a histogram plot of the waiting times for the four-millisecond tasks (only greater than 800 microseconds) for 6000 rpm

Let X be the random variable that measures the waiting times. The goal was to find the probability that P(X>4 milliseconds). The approach taken here is statistical in the sense that a distribution was fitted to the data and then the probability was derived based on the fitted parameters of the distribution.

The software team fitted two distributions to this truncated dataset:

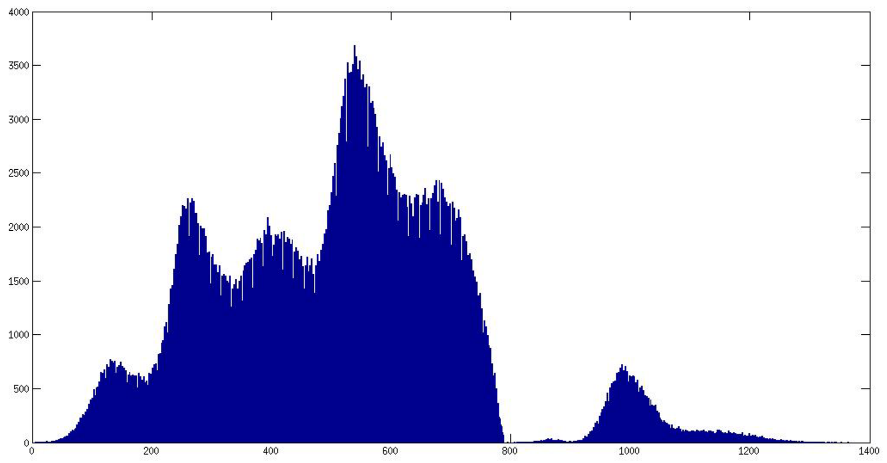
- **Gaussian Distribution.** The estimated mean and variance of this distribution are 982.6 microseconds and 45.1 microseconds, respectively, with 95% confidence in the estimation according to MATLAB's maximum likelihood operation. Then:
  P(X>4 microseconds) = 1 – P(X<4000 microseconds) = 0 (by MATLAB)
  Using the natural logarithm scale (where z is the standard normal variation):
  $P(z) = 1/\mathrm{sqrt}(2pi) * \exp(-z^2/2) ==> \ln(P(z)) = \ln(1/\mathrm{sqrt}(2pi)) - z^2/2$
  For us, z ~= 67.
  Thus:
  $\ln(P(67)) = -2.2454*10^3$
  And hence:
  $P(X=4000) = \exp(-2.2454*10^3)$
- **Generalized Extreme Value Distribution.** GEV distributions are used to model extreme values. GEV combines three distributions (Gumbel, Frechet, and Weibull) into one. It has three parameters: shape, location and scale. MATLAB's statistics toolbox was used to fit a

- GEV to the truncated dataset. The estimated parameters were: -0.1645 (shape), 964.29 (location) and 45.6141 (scale). Based on the value of the shape parameter, it can be inferred that it is a Weibull distribution. Plugging these values into the expression for the cumulative distribution function of GEV, the probability of P(X>4000) = 1-exp(1.1620*10^6) is obtained.

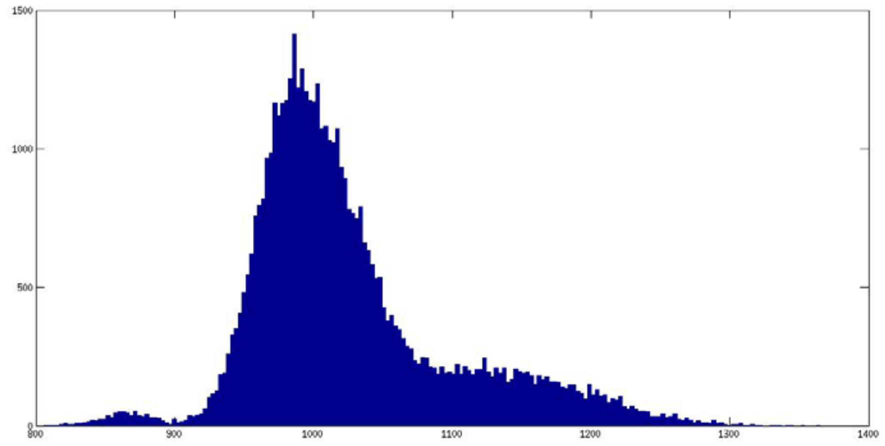
A.14.9 Analysis of 8000 RPM Data

As before, the following histograms show the waiting times for the jobs. The number of events is much higher than the 6000 rpm case. Also, the tail of the distribution spreads more towards the four-millisecond region.

The following histogram plots of the waiting times for the four-millisecond tasks for 8000 rpm.



The following histogram plots of the waiting times for the four-millisecond tasks (only greater than 800 microseconds) for 8000 rpm.



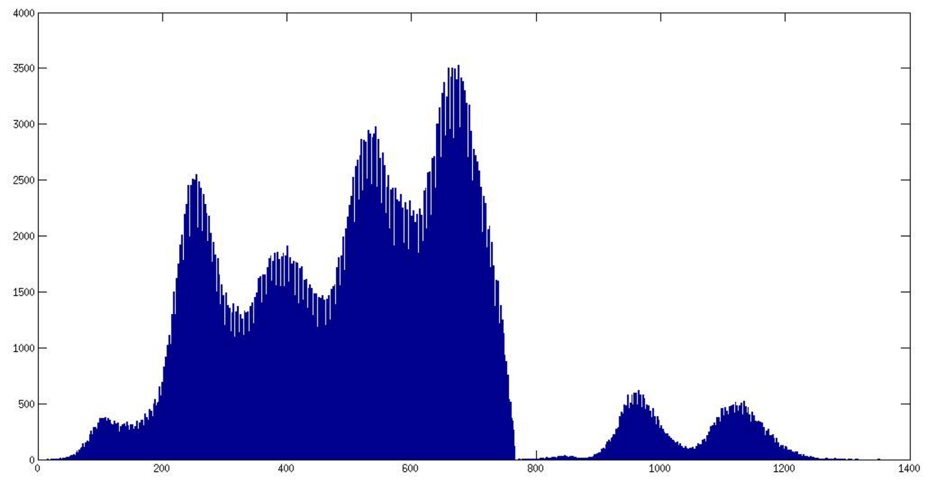
Both Gaussian and GEV distributions were fitted to this data. For the normal distribution, the estimated parameters were 1026 (mean) and 73.5793 (variance), which means:
P(X>4000) = exp(-817.7658)
For the GEV, the parameters were -0.0536 (shape), 995.43 (location) and 59.85 (scale). For this distribution:
P(X>4 milliseconds) = 1 – exp(1.8007*10^4)
A two-component Gaussian mixture model was also fitted to this data. In that case:
P(X>4000) = exp(-1.2563*10^3) + exp(-1.2563*10^3)

A.14.10 Analysis of 9000 RPM Data

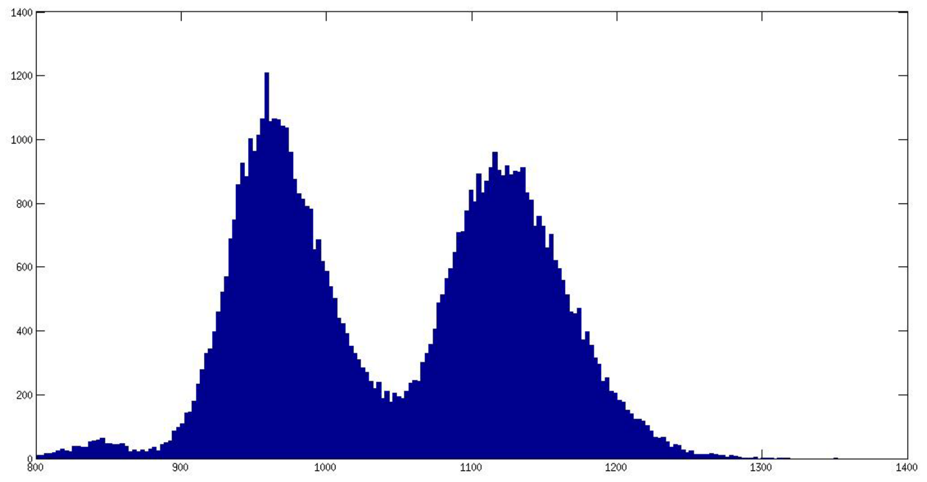
In this case, there are two peaks after the 800 microsecond waiting time. Therefore, fitting a single distribution to this dataset is incorrect. As before, a two-component Gaussian mixture model was fitted and the parameters were estimated using Expectation Maximization. In this case also, the value of:

P(X>4 milliseconds) = exp(-1.2814*10^3) + exp(-1.1473*10^3)

The following histogram plots of the waiting times for the four-millisecond tasks for 9000 rpm.



The following histogram plots of the waiting times for the four-millisecond tasks (only greater than 800 microseconds) for 9000 rpm.



A.15 Worst-Case Execution Time (WCET) Analysis

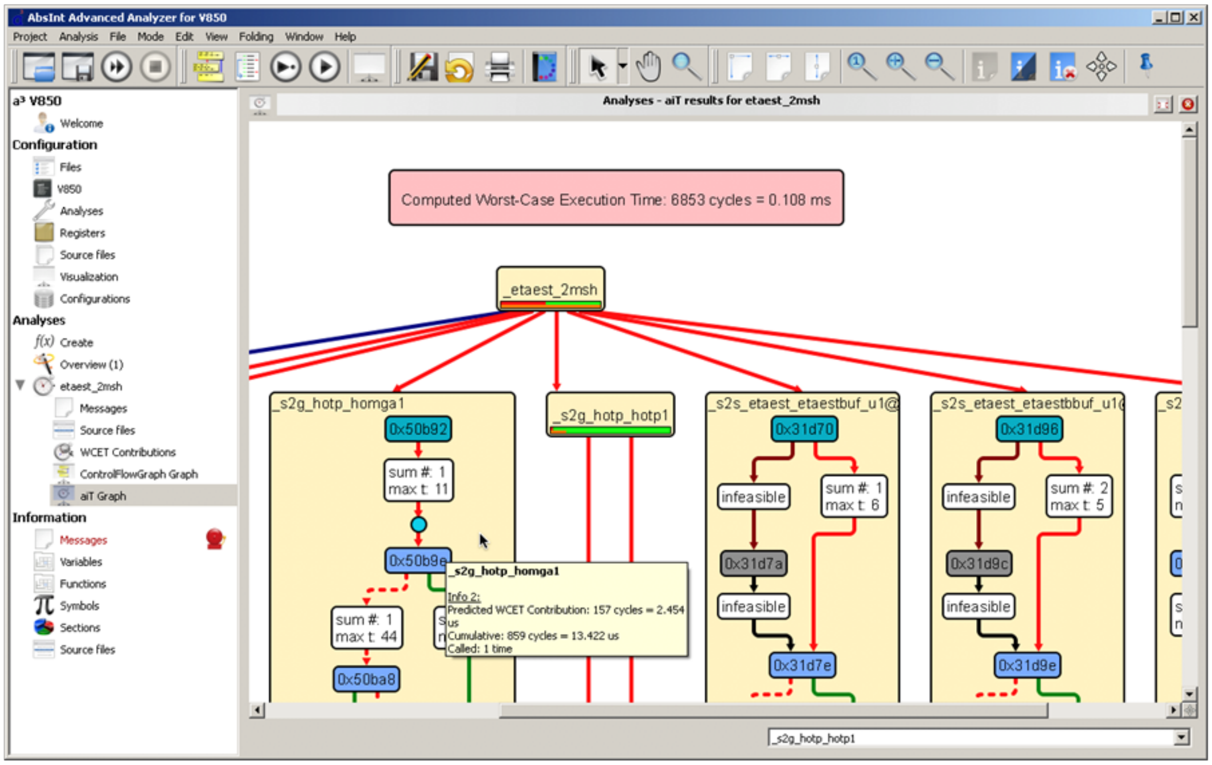
To complement the statistical analysis, a static analysis was performed on the source code itself. This approach to CPU load verification, which involves no runtime measurement of any kind, is known as *worst-case execution time analysis*, or simply WCET analysis. It is a rigorous approach that places a hard upper bound on the execution time of a given software task. The idea is to make timeliness a property that can be formally analyzed.

A.15.1 Static Analysis with aiT

The purpose of this analysis, therefore, is to verify temporal behavior by means of the WCET approach. The WCET tool can provide evidence that a function may not meet its timing requirement during some worst case execution time.

To perform this analysis, the WCET tool needed to simulate the computer hardware and compute the execution of the software on the hardware. The Toyota software executed on a variant of the V850 processor.

aiT from AbsInt supported V850-compatible analyzers and was selected for this study. It was configured, with assistance from Toyota, according to the specifics of the ETCS-i, including stack address, peripheral areas, memory wait states, and other attributes of the hardware. In theory, the tool should then be ready to begin its analysis. The user simply provides an executable containing V850 code, selects the entry point of a task in the executable, and then clicks a button to compute the WCET of the task (see Figure A.15-1). In practice, of course, there are significant limitations.



Computed Worst-Case Execution Time: 6853 cycles = 0.108 ms

_s2g_hotp_homga1

Info 2:
Predicted WCET Contribution: 157 cycles = 2.454 us
Cumulative: 859 cycles = 13.422 us
Called: 1 time

Figure A.15-1. Screenshot of aiT computing the WCET of an ETCS-i function

First, because the ETCS-i has a proprietary CPU, the exact details of which are considered by Toyota to be a trade secret, aiT's model of the CPU is inaccurate. Specifically, aiT's knowledge of the processor pipeline and memory latencies do not match the ETCS-i processor precisely.

As an alternative, aiT provides models of several of the standard V850 variants, and the closest match of these was selected for the WCET analysis. Using this "close" model of the processor meant that the analysis results would not be exact; however, the variation is unlikely to be greater than 20% (in either direction). This variation is accounted for in the analysis results, as discussed in the following section.

Second, there are restrictions on the kind of code that aiT is able to analyze. For example, aiT cannot handle:

- Multitasking
- Interrupts
- DMA routines
- Assembly code

The limitation on multitasking is of particular importance because of the highly multitasking nature of the ETCS-i. If, for instance, a function has a 100-microsecond WCET and runs in a one-millisecond periodic task, there is no danger of overrun, but if ten such functions execute in the same period, then suddenly there is a very real possibility of missed deadlines. aiT is not capable of detecting this vulnerability.

It should be noted, however, that all WCET analyzers, including the most advanced research prototypes, share similar limitations. Overall, aiT is adequate for single-task analysis and is arguably the best in its class given the current state-of-the-art.

A.15.2 Analysis Results

Even with a perfect WCET analysis tool, there remains the task of locating time-critical functions in the ETCS-i code, and also pairing these functions with a predicted time bound (i.e., a deadline as defined by Toyota's specification). This proved more difficult and time-consuming than expected. The scheduling of tasks in the ETCS-i is essentially dynamic, and there are no explicit declarations of deadlines.

The design of the ETCS-i software does have implied deadlines, however, as tasks are scheduled periodically. Therefore, a task scheduled for a one-millisecond period naturally has an implicit one-millisecond deadline.

With this guideline in mind, a sample of periodic tasks was chosen. The entry points (functions) of these tasks were then loaded into aiT for WCET analysis. However, not one of the functions among this selection could be analyzed automatically. Each required special assistance from the user to complete the analysis. For example:

- `gesgm2drv_2msh` (two-millisecond period) has a busy-wait loop whose bound is determined by a timer value. The WCET of this type of loop cannot be derived automatically, and therefore a manual annotation had to be inserted to inform aiT that the loop's WCET was five microseconds.
- `etaest_2msh` (two-millisecond period) invokes assembly code that contains a loop. Because assembly has no fixed code patterns, aiT is unable to positively recognize its structure and thus cannot determine the loop's bound. According to Toyota engineers, however, the loop should iterate no more than 20 times, and this fact was expressed to aiT as an annotation, allowing the analysis to complete.
- `gehdlp_req_1ms` (one-millisecond period) contains indirect recursion in its call path. Like most static analysis tools, aiT cannot handle this type of recursion without manual annotations. The exact parameters for these annotations could not be derived from the

Toyota specification (see section "Recursion") and as a result, this function was left unanalyzed.

For every instance in which aiT was able to find the WCET, it was consistently a small fraction (always less than 10%) of its corresponding period. For instance, the function `gesgm2drv_2msh` (two-millisecond period) had a WCET of 49 microseconds. Even when taking into account the fact that aiT is employing a different CPU model for its analysis, a 20% higher WCET would still lie well within the required parameters.

To be clear, not all of the periodic functions were analyzed. Having to assist the tool with manual annotations on every function placed a limit on the amount of work that could be done. However, the set of functions that were successfully analyzed are likely a representative sample of the whole.

A.15.3 Recursion

During the WCET analysis effort, the discovery of recursion presented a problem, as mentioned in the above description of the `gehdlp_req_1ms` function.

Recursion occurs when a function calls itself, either directly or indirectly. The direct case is far more common and is typically known as *self-recursion*. Certain well-known algorithms, such as finding Fibonacci numbers, can be expressed elegantly using self-recursion.

The indirect case occurs when, for example, function A calls function B, B calls C, and C calls A. This type of recursion is exceedingly rare, for the following reasons:

- The abort condition of the recursive ring usually cannot be detected automatically, and therefore automated analysis tools (such as aiT) are unable to handle them. This prohibits the kind of verification and validation that is necessary for safety-critical and hard real-time systems like the ETCS-i.
- Given that the call structure of indirect recursion is too complex for automated tools to handle, it is typically too complex for human programmers to fully comprehend, as well. This complexity makes a manual analysis ineffective.
- Deeply nested recursion could exhaust the stack space, leading to memory corruption and run-time failures that may be difficult to detect in testing.
- Recursion is essentially a loop, and like any loop, there is a danger of non-termination. An infinite loop could lead to deadlock that results in a system reset.

The question, then, is how to verify that the indirect recursion present in the ETCS-i does in fact terminate (i.e., has no infinite recursion) and does not cause a stack overflow.

A.15.3.1 Toyota's Recursion Analysis

Toyota's response to this question is that a manual code review, followed by extensive field testing, has not shown any evidence of software quality problems related to the recursion. If, for example, an infinite loop had occurred, the watchdog timer would initiate a system reset, but a reset never occurred during testing.

For the case of stack overflow, the CPU in the ETCS-i does not have protected memory, and therefore a stack overflow condition cannot be detected precisely. It is likely, however, that overflow would cause some form of memory corruption, which would in turn cause some bad behavior that would then cause a watchdog timer reset. Toyota relies on this assumption to claim that stack overflow does not occur because no reset occurred during testing.

Toyota used a tool called gstack from Green Hills Software to determine maximum stack size requirements. The tool gstack is a static analyzer that computes an upper bound on the stack size required by a given executable. For the ETCS-i, it reported the maximum possible stack size as 1688 bytes. This result comes with a caveat, however: gstack cannot account for recursion, as stated in its user manual:

> *gstack cannot work if there are potential direct or indirect recursive calls in the program because it cannot predict how many times the recursion can occur. gstack prints a warning message if it detects a possible recursion in the call graph.*

Faced with this limitation, Toyota added an extra margin of safety to the predicted bound by allocating 4096 bytes for the ETCS-i stack—more than double the original prediction.

As to why recursion was present in the ETCS-i software, Toyota reported that it was a deliberate design choice in order to simplify the quality assurance process and reduce the total size of the executable. The recursion made this possible by allowing part of a newly implemented state machine to be linked to a state machine that was already present in the code. (The `gehdlp_req_1ms` function is just one of three sites where recursion is present.) This linkage allowed existing code to be reused, unmodified, and therefore did not require additional testing nor contribute to an increase in code size.

A.15.3.2 Recursion Analysis

Toyota re-examined the state machine code where the recursion occurred. The company emphasized a high degree of confidence in its analysis, stating that the recursion is bounded and contains no infinite loops, no dead-end states, and no stack overflow conditions. In particular, the company reported that a recursion can occur no more than once for any given code path.
It should be noted that the ETCS-i state machines were implemented by hand, not generated from a model, and therefore a manual code review was the only analysis option available to Toyota's engineers.

Moreover, a fully automated analysis is not possible, given that the presence of recursion defeats stack usage tools such as *gstack* and the *AbsInt StackAnalyzer*. To support the analysis, a partially automated, tool-assisted approach was therefore taken. It involved a C code analyzer called *ncc*, which can pinpoint recursion sites and produce accurate call graphs of the ETCS-i code, as shown in Figure A.15-2.

Figure A.15-2 shows a portion of the `gehdlp_req_1ms` call graph, pruned of all nodes that do not participate in the recursion. (All function names have been obfuscated, without loss of structure, to protect Toyota's intellectual property.)

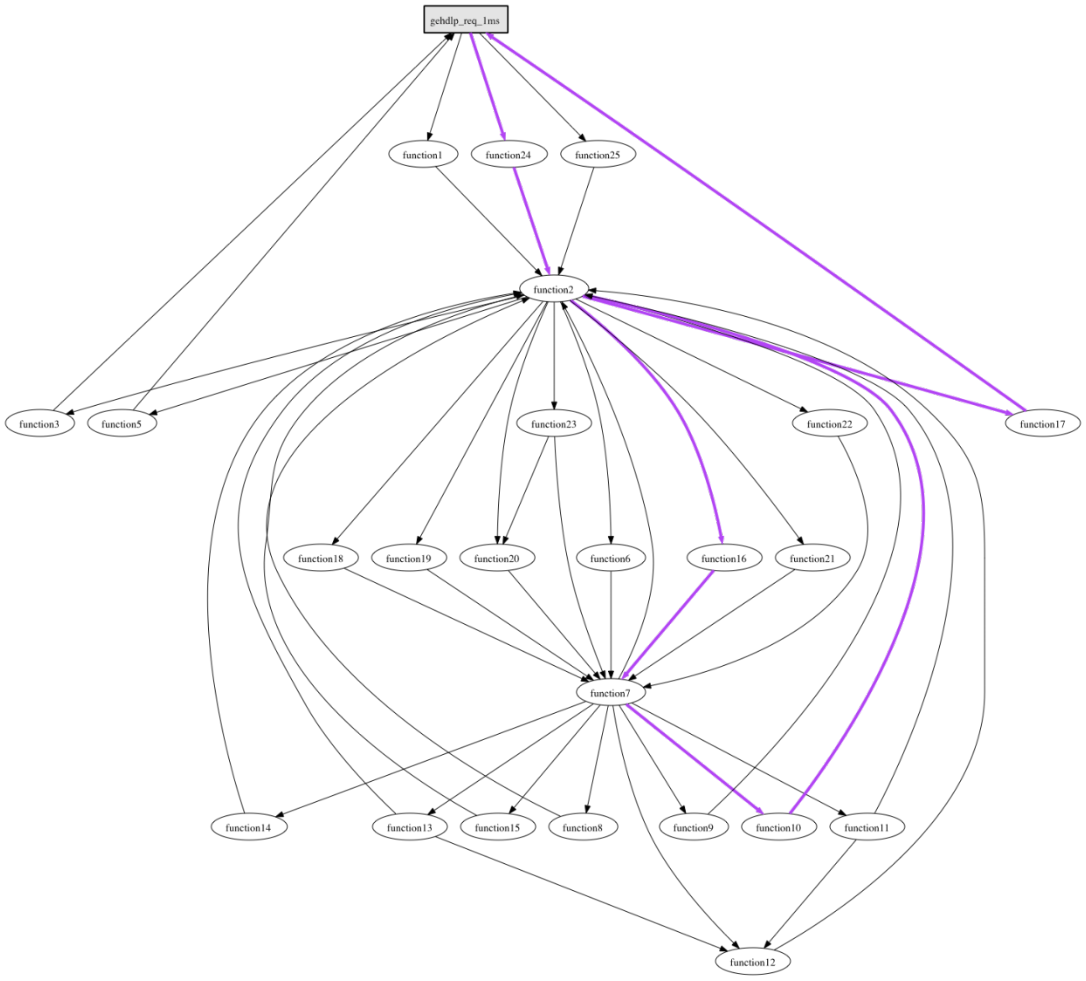


Figure A.15-2. Portion of the `gehdlp_req_1ms` Call Graph

One immediate observation from this figure is the sheer number of possible paths the graph contains, even after eliminating all non-looping nodes. (In fact, there is an infinite number of possible paths due to the presence of cycles, thus demonstrating why analysis of recursive code is so problematic.) A complete manual analysis would be impossible, although the visual aid can make a partial analysis somewhat simpler.

By referring to the call graph, it is possible to trace the flow of the state machine implementation, which is essentially a table lookup to determine the next state. A manual (static) analysis cannot, in general, determine what this state will be at run-time, but in the special case of `gehdlp_req_1ms`, the state machine tables happen to be composed in a way that the program flow is deterministic. For example, at some points in the program flow, the next state is always a constant, regardless of the current state.

Based on this knowledge, the exact program flow for `gehdlp_req_1ms` can be determined. It is indicated by the thick purple lines in Figure A.15.-2. The flow reveals that the function is actually *doubly* recursive: It flows from `gehdlp_req_1ms` to `function2`, recurses back to `function2`, and then recurses again back to `gehdlp_req_1ms`.
Toyota engineers confirmed this software recursion.

It is not clear what impact recursion has with respect to the larger UA problem. Whether one recursive loop or two, there are other sites of recursion in the ETCS-i that remain unanalyzed.

A.15.3.3 Recursion and System Reset

The WCET analysis and recursion analysis involve two distinctly different problems, but they have one thing in common: Both of their failure modes would result in a CPU reset. In the WCET case, there is nothing in the design of the ETCS-i that would necessarily prevent a deadline overrun from occurring. But if such a problem does occur, and the event queue fills up, the operating system will detect this problem and kill the watchdog timer, which will in turn cause a system reset. Likewise, an infinite loop due to bad recursion would also cause a watchdog-generated reset, and possibly a stack overflow (unconfirmed).

These potential malfunctions, and many others such as concurrency deadlocks and CPU starvation, would eventually manifest as a spontaneous system reset.

A.15.3.4 Timing Analysis Results

The timing analysis identified no timing influencing the onset of any UA event. It is highly likely that a starvation- or recursion-related malfunction would lead to a system reset.

A.15.3.5 Timing Analysis Forward Work

Concluding the full analysis would require the following work:

- **Statistical Analysis.** The statistical model shows a low probability of CPU reset due to computational starvation based on an expected billion hours of driving per annum of the relevant Camry fleet. This analysis is as detailed as time permitted. Future work would involve a more detailed analysis that combines WCET analysis with a statistical modeling of worst-case task interleaving.
- **WCET.** A more accurate hardware model, as well as performing WCET analysis on a larger sample of software functions, would provide further evidence for system performance. This could be combined with a statistical model of worst-case task interleaving to place an upper bound on total CPU load.
- **Recursion.** To perform automated analysis of the hard real-time behavior, complete removal of recursion from the software would be required.
- **System Reset.** Homework #116 contains data from a single test Toyota performed. This study has not conducted sufficient hardware-in-the-loop testing to determine the behavior of the system under spontaneous reset conditions. Additional field testing to observe the acceleration behavior of the system during resets could be conducted. The purpose would be twofold: Demonstrate benign behavior of all control functions under both single-reset and multiple-reset (cascading) conditions. (Alternatively, the latter could be accomplished through fault tree analysis instead of testing.)

A.16 Green-Hills Executable based Testing

Developmental build of the entire Toyota code base (denoted as the Green-Hills Executable), on the Green Hills RTOS development environment (MULTI) began relatively late in the study (mid July) and therefore, due to the size of the code base, can only provide intermediate results. The advantage of the Green-Hills Executable approach is its capability of executing the entire real Toyota/Denso code, such as the entire Main CPU block, a large body of code, consisting of more than 1700 modules. This is a complex code base that performs workflow all the way from pedal and throttles inputs, through pedal and throttle learning, damping, idle speed control, cruise control throttle decisions, and various other components. The workflow ends with the output throttle command sent to the throttle motor.

The purpose of this Green-Hills executable approach allows the discovery of pure software bugs, including those that emerge only in conjunction with simulated hardware faults, all while using high throughput automatic test generation and automatic computer aided property checking.

A.16.1 Technical Approach

The Green-Hills executable build was performed using the same object files used by Toyota/Denso on the actual 2005 L4 Camry vehicle. The only exception is that the object files were not linked with the OSEK RTOS. Because critical Main CPU tasks are invoked from a 4ms A/D interrupt handler using a fixed schedule, a fixed scheduler was implemented. This implementation eliminated the need to incorporate OSEK into the Green-Hills executable build. Most task are 4ms tasks, with a small number of other tasks being invoked on a 8ms or slower time schedule. The software team used simulation time, in the form of counters, similar to Toyota's implementation on the vehicle, to invoke the tasks.

A.16.2 Current Status

The Main CPU build of the Green-Hills executable was completed. However, several sanity tests using data provided by Denso to assure that it is executing properly were needed. The existing Green-Hills executable was used to generate pedal and throttle diagnostic maps for the purpose of assuring the accuracy of the team's Matlab models. This testing used the Toyota source code to verify the model accuracy. The source code generated diagnostic maps matched the model generated diagnostic maps point by point.

A.16.3 Verification Issues

The Toyota software testing approach consists of unit tests, where units are individual modules, which in turn correspond to tasks, many of which being periodic tasks. From discussions with Toyota engineers, it was understood that a software test harness for the entire code does not exist. Absent an overall software test framework prevents the use of high volume computer aided verification techniques, such as automatic test generation.