Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

An Approach to Injecting faults into Hardened Software

Dave Edwards, Lori Matassa
Intel Corporation

Abstract

There are many efforts within the Linux* community to produce a distribution of Linux* that meets industry standards for quality and reliability. There has been acknowledgment for the need to introduce faults into various software layers of the Linux* OS to achieve this. This paper focuses on the results of our development of a prototype fault injection harness. The prototype focused on a black box approach for injecting faults into device drivers. The technology proved in this prototype can be applied to any software layer in the operating system. This presentation proposes and proves the feasibility of a method for injecting faults called, “state analysis.” This method is the key to our black box approach for driver hardening. It does not require a test writer to have intimate knowledge of the implementation for the driver. It also provides a solid foundation for driver developers to augment the fault injection harness to meet whatever the Linux community presents to the world in the way of driver hardening criteria. The target audience includes developers focusing on Linux* hardening (Drivers and Kernel), test engineers looking for a starting point to fault injection, and anyone looking for input into the kinds of capabilities that can be provided by the use of fault injection.

1 Introduction

A hardware device has a finite set of functions to perform and a rigid programmatic method for utilizing its functions. A device driver con-

tains many code paths that exercise the functionality of a hardware device. This paper discusses the learning obtained from a prototype fault injection test harness in which hardware faults are emulated and injected into Linux* device drivers.

1.1 Purpose and Scope

The purpose of this paper is to provide insight into fault injection through the discussion of a prototype fault injection harness implementation. This paper and will provide its audience with one proposed method for ensuring the hardening level of a device driver.

Actual design details of the entire prototyped implementation are not included in this document.

1.2 Intended Audience

This paper is intended for development and test engineers and anyone interested in designing, implementing or utilizing fault injection capabilities that are reproducible, portable across software revisions and flexible.

1.3 Recommended Reading

The appendix of this paper contains background and reference information. This information is provided to assist in clarifying concepts that are touched upon in this document. It is recommended that these be looked at closely once the basic concepts are understood. It is expected that these sections will provide sufficient detail to explain anything that has not been directly addressed in the main portion of this paper.

The overview sections contain information about the purpose behind fault injection and how each of the software components are related with regards to their interfaces.

The section titled, “Fault Injection (FI) Prototype” describes each major component of the prototype in a little more detail. The purpose of which is to describe the intent and major function of each sub-component.

Definition of Terms

State: A deterministic path from one starting point to another. A state refers to the various states of the hardware as it is programmed (by a driver) for its particular function.

State Analysis: The process of tracking the state of hardware and making decisions about what to do as various hardware states are encountered.

State Machine: The mechanism that can track and respond to changes in hardware state. The machine itself consists of a collection of code segments.

Code Segment: A simple code fragment that provides the functionality of the state machine.

State Machine Test: This is the input file used by the State Machine Compiler to create a binary state table that can be dynamically loaded.

FI Engine: Fault Injection Engine. The concept of an engine refers to the central control component for monitoring state and injecting faults into a device driver.

2 Driver Hardening Overview

Device drivers can be a source of operating system instability and are often contributors to system degradation and/or unscheduled outages. Therefore, device drivers must be robust. A hardened device driver is a robust device driver. Hardened device drivers are designed and developed with the focus of minimizing the instability and downtime of the system.

Measuring the hardness of a driver is difficult and unclear. A concept known as driver hardening levels is documented in a white paper titled, "*Device Driver Hardening and Manageability*." The white paper can be found at the **http://developer.intel.com** web-site. These levels are used to define fundamental hardened driver guidelines, measure the hardness of the driver and create a better understanding as to how robust a driver is. These guidelines are used by device driver writers who wish to support higher levels of availability through the use of some or all of the hardening techniques described in each level. The levels include:

Level 1 - Stability and Reliability Includes good coding practices and requires fault injection testing.

Level 2 - Manageability Provides information that can be used by driver management applications to understand the status of the system and to identify potential problems that might be growing. This information includes driver statistics, event logging and driver diagnostics. All of this information is essential in proactively recognizing potential problems. Together, this information can identify a problem and report it immediately so that downtime can be prevented or at least minimized. Therefore, handling the fault gracefully.

Level 3 - High Availability This is the highest level of a hardened driver. High availability systems minimize system downtime. Guidelines in this level support high availability features which enable a driver to repair or reconfigure devices without needing to power down or reboot the system. These guidelines also include fault recovery to the extent that when a fault is identified, the driver repairs the fault if it can keep the device in service and, at a minimum, isolates the operating system from being affected.

Fault injection can be applied to any level of driver hardening. for illustration purposes, this paper focuses on level 1 where a developer is ensuring the integrity of the source code through fault injection, before introducing the work effort to validation.

For purposes of this paper, this paper concentrates on Level 1 Hardening. Level 1 Hardening guidelines specify that hardened drivers must be fault injected tested.

Good coding practices alone cannot ensure the stability and reliability of a system. Device drivers typically are written and tested with emphasis on the normal operation of the hardware. Details as to how a driver identifies and recovers from faulty hardware or system conditions are often minimal.

Hardened device drivers are designed to be more robust because they are coded to expect anomalies and process them in a way that minimizes the impact to the overall system, thus preventing unplanned downtime of the system. The implementation of the code should test for such things as: values that are illegal, states that should never occur and expect that the device should complete a command within a defined amount of time.

The only way to test a driver's robustness is to include tests that purposefully inject conditions that simulate hardware and system faults. This is known as Fault Injection Testing.

There are several ways to inject faults into a system. The most common method involves altering the branch paths to purposefully modify good data into bad data. This can be accomplished with many tools. For instance, the use of in-circuit emulators (ICE's) or in-target probes (ITP's) can change the execution path and data values at run-time. Other methods include the use of debuggers or special code additions with the specific purpose of causing error paths to be exercised. This is known as "white box" testing where the goal is to maximize code coverage.

White box tests pinpoint exact areas and values within the software that are changed. This means that the test is implemented knowing exactly where and what will occur. There are a few issues with white box testing. First, the setup for the test is very labor intensive and in some cases, requires complete manual intervention to control and execute tests, thus making it nearly impossible to repeat test results consistently. Second, there is the possibility that the object code under test is not the same object code that is shipped as the final product. This is not acceptable to most suppliers of high availability and hardened systems.

There is a method of implementing fault injection tests that is fully automated, can provide reproducibility in test results and uses the same version of object code for fault injection testing as the shipped product. This method, known as "black box" testing. Black box testing uses carefully designed tests that emulate faults at software layers below the component under test. In the case of the prototype fault injection effort, faults are emulated in the hardware layer and the component under test is a hardened device driver.

This method for injecting faults that can be automated, provides reproducible test results, is portable across driver revisions, and is simple to augment as software capabilities and test requirements change. This concludes the overview of driver hardening, why fault injection testing is required and different approaches testing. The remainder of this document will describe the key concepts and critical design details for a prototype implementation.

3 Software Overview

There are three main components to the prototype, the System Driver, the Fault Injection Engine and the Common Driver Interface (hooks). The system driver is the component under test that utilizes a Common Driver Interface (CDI) that contains software hooks to interface with the FI Engine component transparently. The FI Engine is the core software component, providing all of the services necessary to inject faults into a device driver.

Device drivers execute in kernel space. As such, the FI Engine is a driver with interfaces designed to allow connections to occur between a system driver and itself. The CDI (used by the system driver) contains special software hooks that allow it to connect to the FI Engine during the driver initialization sequence. Once the connection is made there is a method to make a direct call to the FI Engine from the connected system driver. For the prototype, the CDI consisted of macros that were created to represent an abstraction on the existing Linux* macro set for programmed IO, DMA and PCI configuration.

State analysis is the process of tracking I/O

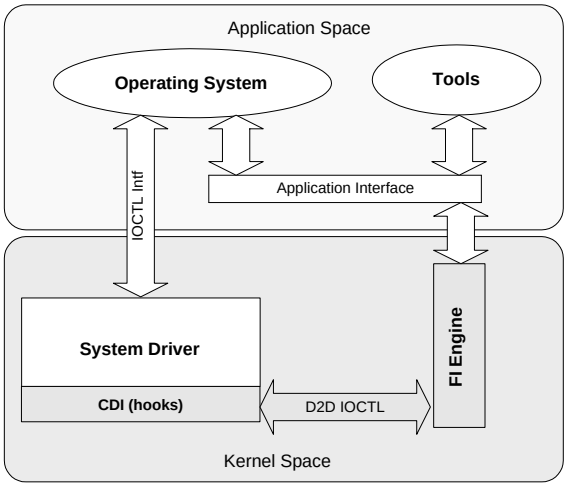

Figure 1: Major Software Components

transactions for the purpose of 1) detecting hardening violations 2) injecting faults at specific points in the usage model for that hardware and 3) to emulate the appropriate hardware behavior from the time a fault is injected to the time the hardware is expected to be executing normally.

Hardening violations are categorized by warnings, and rule violations. A warning is some indication that a driver inappropriately accessed registers given the current state of the hardware. This can be useful for detecting known hardware problems that have potential for causing failures under adverse conditions, but aren't guaranteed to do so every time. This is a way for a test engineer to create warning flags for special events. A hardening violation is one in which a driver responds to a failure in such a way that it is known to be inappropriate. For example, a driver may not clear interrupts within a control register after a given fault. In these situations, it is thought that the driver will cause a system failure either immediately, or shortly thereafter.

The process of writing a test begins by using data sheets and any other hardware documentation that specifies where the hardware can fail. Once hardware failures are understood, a test writer can create fault scenarios in which the hardware could fail during its normal operation. These scenarios are then translated to state machine form in which the state of the hardware is tracked, and at various points a decision can be made to inject a fault between state transitions. Once failure scenarios are translated to a state machine form the test writer can, compile the test and load the test into the FI engine before the system driver is loaded.

Figure 2: "State Machine Capabilities" illustrates the services that a state machine engine provides and the types of information that a test writer must have to track the state of hardware, inject faults, and emulate hardware fault behavior.

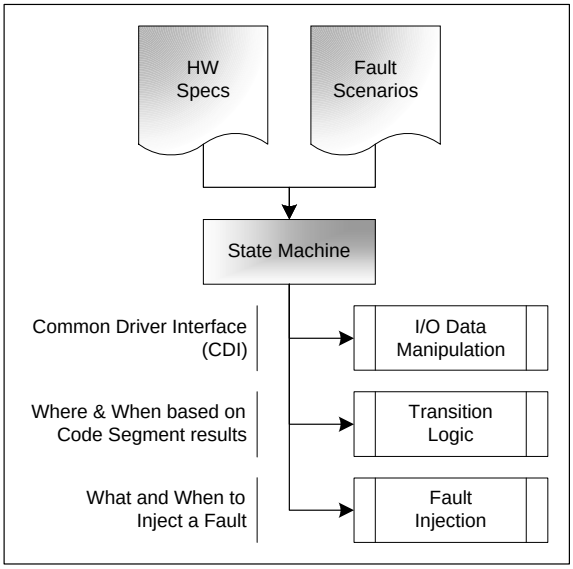

Figure 2: State Machine Capabilities

Figure 2 also illustrates the three main components of the state machine, I/O Data Manipulation, Transition Logic and Fault Injection capabilities. I/O Data Manipulation consists of

passing the CDI input parameters to the FI engine and allowing the engine to use the values to determine state and to manipulate the values, i.e. the hooks. Transition Logic consists of a method for executing code associated with a state and being able to specify which state to transition to. This could also be considered execution flow control. The Fault Injection piece defines the method in which the test writer specifies exactly when I/O Data is to be manipulated and how the FI engine will respond for subsequent calls to the CDI.

When all of these things are used properly, a software engineer can track the state of a hardware device. This allows them to specify exactly when to inject a fault and how to behave once the fault is injected.

4 Fault Injection (FI) Prototype

The prototype consists of a system driver, state machine compiler, and an FI engine driver. The core technology described in this paper is the state analysis component of the FI engine. It is responsible for the state tracking and fault injection capabilities. Figure 1: "Major Software Components" outlines the relationship between all of these components which are described in the following sections.

4.1 System Driver

The System Driver (illustrated in Figure 3) is built using the CDI to access hardware resources. The main advantage is that this method gives the engine access to all the possible input and output parameters of the transaction. As such, a test engineer can make decisions and modify any of the data that gets transmitted between the CDI and hardware. In general, execution control is passed to the engine after a read transaction and before a write transaction such that the FIE can decide what to do with the data that will be returned to the driver or written to hardware.

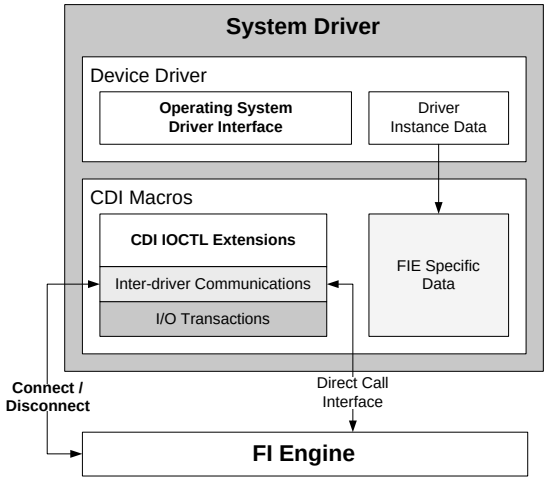

Figure 3: System Driver Block Diagram

The process for initiating fault injection testing involves a dynamic connection sequence. When a system driver is loaded the driver initiates a connection sequence by calling kernel routines that return a handle to the FI Engine driver. Once the handle is known, the system driver initiates driver to driver IOCTL calls through the kernel to the FI Engine. The first call to the FI Engine sends a request to connect. The FI Engine grants this request and returns a handle to a data structure representing instance data for that connection. The instance handle contains the address of the FI Engine entry point function. The CDI uses the entry point address to call directly into the FI Engine.

When a system driver is unloaded, a disconnect sequence is initiated through the same driver to driver IOCTL interface. At which time the FI Engine will perform cleanup on any state machine configuration data associated with the connection.

4.2 Fault Injection Engine

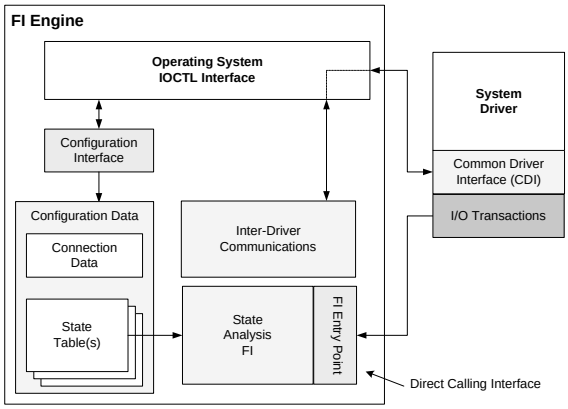

Figure 4: Fault Injection Engine Block Diagram

The FI Engine is illustrated in Figure 4: "Fault Injection Engine Block Diagram." The various blocks within the diagram represent key internal components. Lines are drawn to show the component's inter-relationships. Arrows have also been added to hint at data flow. The system driver connects through special macros of the CDI and is intercepted by the Inter-Driver Communications (IDC) component. The IDC creates data structures and initializes internal subsystems. The Configuration Interface provides an application with the ability to load a state machine table dynamically. The state analysis component is called directly by the CDI once the connection sequence completes.

NOTE: the actual I/O access will occur in the system driver, not the engine. The engine only modifies the data before or after the actual I/O.

Figure 5: "CDI Hooks to FI Engine Flow Diagram" illustrates the flow of execution with respect to the CDI hooks and the FI Engine. It starts with a driver making use of a CDI Macro. Typically an I/O transaction involves either a read or a write. Thus the concept is fairly simple. You inject a fault into a driver by modifying values returned by the read transaction to trick the driver into thinking that a status register contains an error value. You can also inject a fault into a driver by modifying data from a write transaction before the data is transmitted through the I/O interface.

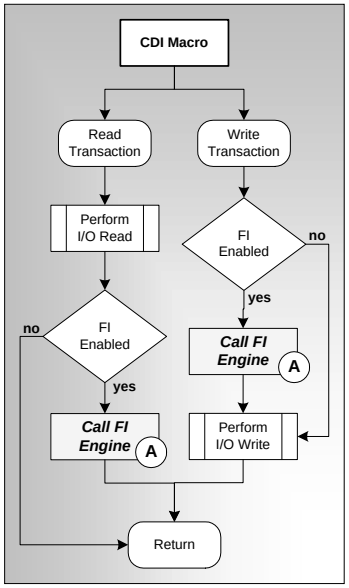

Figure 5: CDI Hooks to FI Engine Flow Diagram

Once the FI Engine receives execution control, it begins traversing the state table, executing code segments that are related to the current state of the hardware. On entry to the engine, the state machine determines where it left off the last time the engine was called. When a driver first connects, the starting state will be the very first state of the state table. The state machine parses the state table entry for the code segment and then executes code associated with that code segment. The return value of the code segment is then used to determine which state should be traversed to. This will continue until the ExitStateMachine code segment is executed. This is a special piece of code that allows a test writer to specify where the state machine will continue the next time it is called and exits the FI Engine to allow the

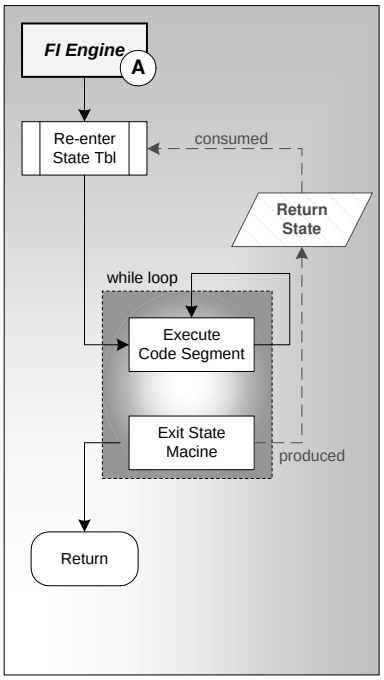

Figure 6: FI Engine Flow Diagram

driver to continue running.

4.3 FI Compiler

The compiler (Figure 7: FI Compiler Component Diagram) produces a binary output file that is loaded into the engine with a command line tool. The compiled file is translated to a state table and stored for retrieval when the device driver makes a connection to the FI Engine.

Input for the compiler consists of a state machine test file and a code segment definition file. The state machine test file contains "source" text that is translated to binary form by the state machine compiler. The code segment definition file is created from the code segment definition data structure mentioned in the next section. The code segment definition file is created as part of the build process. When a test is compiled the code segment definition file is used to validate the input parameters in the test being compiled.

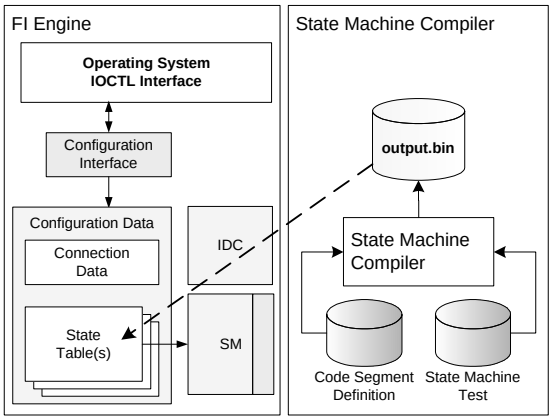

Figure 7: FI Compiler Component Diagram

4.4 State Machine

The overview section of this paper made claims that this prototype proves the feasibility of creating reproducible test results, portability across driver revisions, and simple augmentation of the state machine. The first portion of this section is dedicated to explaining why it can do these things and the remainder gives insight into how it can do them.

The state machine is central to the reproducibility of test results consistently across software revisions. It does this by allowing a test developer to track the state of the hardware and specify the exact moments in which a fault will be injected. By tracking hardware state, the test developer does not have to tie the test to the implementation of the driver, only the implementation of the hardware. Thus, as long as the hardware fault scenarios don't change, neither does the test, for any revision of the driver.

The state machine is designed to be augmented as test requirements and driver design dictate

the need for change. There are three supporting components to the state machine, the state machine switch statement, the state table, and the code segment definition structure. These will be described a bit more, shortly. A change to the machine is a change only to each of these three items.

The state analysis component of the FI Engine (Figure 4) contains the state machine. The state machine is responsible for traversing a list of states within a state table. At each state transition, a code segment is executed, which can be considered analogous to a CPU executing an instruction. Code segments are simple, small code fragments that do not depend on one another to complete execution. However, they do have a mechanism for passing data to between states. For the prototype, this was done through a special Reverse Polish Notation (RPN) based stack.

The state machine is a while(1) loop with a big switch statement inside. Code segments are the case statements with the associated code to be executed on a transition into the state. Each state table entry contains a list of states to transition to, based on the return value from the code segment. As illustrated in the Figure 8: "Sample State Table Linkage," the next state list is an array where the next state is determined by using the return value as the index to the array. The next state is pointed to by the contents of the value indexed in the array. The code sample at the end of this section illustrates this process. The end result is very low overhead between executing code segments.

Figure 8 also shows a sample diagram representing the contents of a state table and how the flow is controlled. The first field is the code segment tag; in this case there are four code segments, IsPioRead, OrData, LogWarning and ExitStateMachine. These would be used by the test engineer to create a state machine test.

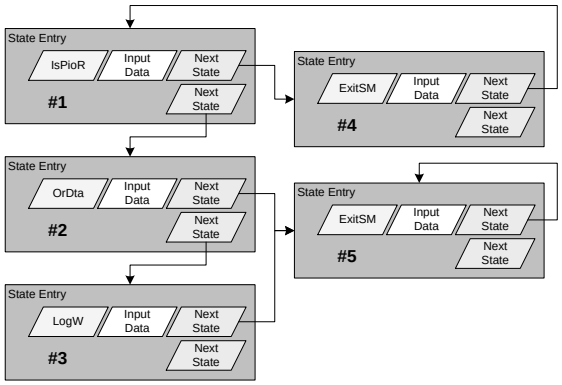

Figure 8: Sample State Table Linkage

In order to extend the state machine, a data structure that defines the code segments is created. Each table entry contains the following information:

String: The reserved word for the code segment.

Number of Inputs: Number of data parameters required for input.

Number of Returns: Number of possible return values.

This structure is shared between the state machine and the compiler. The compiler uses it in the form of a file (saved to disk by the build process) called the code segment definition. The compiler can load the structure to validate the syntax of the test source code, created by the test developer. The state machine engine uses the code segment definition structure to traverse the state table and access data input parameters. A typical code segment definition would look like the following:

Reserved Word	Inputs	Returns
"PioRead",	0	2
"OrData",	1	1
"LogWarning",	0	1
"ExitMachine",	1	1

The following is an example of the state machine implementation referenced in Figure 8, and its associated code segments. See Appendix B for a complete, functional state machine example.

```
FIE_EntryPoint(InstanceHandle *handle) {
   StateTableEntry *this_state = handle->CurrentStTblEntry;
   CdiParameters   *cdi_param  = handle->CdiParam;
   BOOL             exit_flag = FALSE;

   while (exit_flag == FALSE) {
      switch(this_state->CodeSegment) {
         case CODE_SEG_IsPioRead:
            cs_return = 0; /* FALSE */
            if (cdi_param->Transaction == CDI_PIO_READ) {
               cs_return = 1; /* TRUE */
            }
         break;
         case CODE_SEG_AndData:
            cs_return = 0; /* only 1 return */
            cdi_param &= this_state->StateData;
         break;
         case CODE_SEG_LogWarning:
            cs_return = 0; /* only 1 return */
            WriteLog("Warning Violation...");
         break;

         case CODE_SEG_LogViolation:
            cs_return = 0; /* only 1 return */
            WriteLog("Hardening Violation...");
         break;

         case CODE_SEG_ExitStateMachine:
            cs_return = 0; /* only 1 return */
            /* This code segment's data item contains a pointer
             * to the state to execute on the next entry to the
             * state machine.
             */
            handle->CurrentStTblEntry = this_state->StateData;
            exit_flag = TRUE;
         break;
         default: { /* invalid code segment */ };
      }
      /* Set the next code segment to be executed based on the
       * return value of the current code segment.  This is just
       * an array of pointers to state table entries.
      this_state->CodeSegment = this_state->NextState[cs_return];
   } /* end while */
}
```

5 Summary

The prototype fault injection harness proved that it is possible to create a state machine language and that it is possible to track the state of hardware from initialization all the way through the normal execution cycles of a driver.

Tracking the state of hardware is critical to making decisions about when to inject a fault. It is also what allows test results to be repeatable and allows a test to be portable across driver revisions.

Keep in mind that even though this paper focuses on drivers, any software component can make use of the principles.

6 Acknowledgments

Special thanks to my fellow co-workers for their contribution to this effort: Donald Long, Daniel Vanhoozier, Kimberly Davis, Ryan Kummet, Susan Foster, Victoria Genovker, Grace Hawley, and Lori Matassa.

References

[Intel] Lori Matassa *Device Driver Hardening and Manageability* Intel Corporation. `http://developer.intel.com/` (2001)

7 Appendix A - Sample Test

This is a sample state machine test file. The text below describes three major aspects of the system driver under test, normal execution flow, a special IOCTL interface and controlled hardening violations.

A sample driver was designed to illustrate concepts and prove the feasibility of a state machine language. The LCD device is a simple serial port device that plugs into COM1. The sample driver was written to interface to the LCD device. Any programming of the hardware occurs with the serial controller.

The compiled state machine test is loaded into the FI engine prior to loading the LCD driver. When the LCD driver is loaded it makes a connection to the FI Engine. From the moment the driver successfully connects, the state machine is monitoring programmed I/O.

Once initialization completes (successfully) the normal operation of the driver can begin. An application running in the background sends constant messages to the driver through the read/write operating system interface as a character mode driver. The state machine test is designed to inject a fault into the write status register every 1000 character writes to the display. When the driver detects a failure it flashes an error message to the display, re-initializes the device and continues to accept character messages from the background application.

There are some special flags that can be set to demonstrate different aspects of fault injection. The LCD driver has a custom IOCTL interface to allow an application to do three things, re-initialize the driver, set the Warning Violation code path and set the Hardening Violation code path. The background application has the ability to utilize these features on demand.

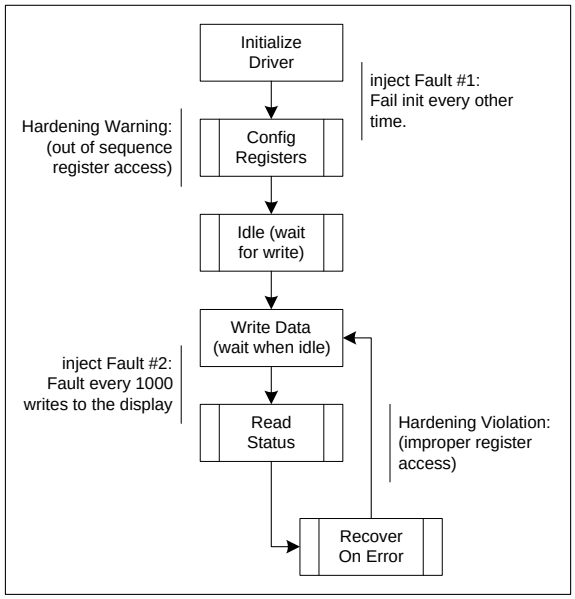

Figure 9: Flow Diagram for the Sample (LCD) Driver

Figure 9: "Flow Diagram for the Sample (LCD) Driver" illustrates the various code paths that can be exercised depending on the run-time switches controlled by the driver's custom IOCTL interface. There are three demonstrations from this, fault injection during driver initialization, hardening warnings, and hardening violations.

There is a run-time switch within the initialization sequence that forces the driver to execute a warning violation when initialization sequence is repeated. The state machine will detect an out of sequence register violation and write a string to the log file. To demonstrate the ability to inject faults during initialization, the state machine is also designed to inject a fault every other time the background application reinitializes the LCD driver.

The final demonstration is to inject a fault during the character write process and force the driver to inappropriately recover from the failure; this is documented as a hardening violation. The driver simply fails to clear the display after detecting a failure and the state machine can detect the absence of that action.

The state machine definition (below) file is fairly complicated to read, but the fact that it can do what it's supposed to, is a major milestone to prove the capabilities of this method of injecting faults and detecting violations. The code segment definitions for the sample machine below, are not defined in this document. The intent is to provide a reference to a working state table and the driver model specified in previous pages.

```
## Comments are permitted in a state definition file if they are
#  preceded by a '\#'.
#
#  The syntax of this state machine test file is as follows
#  (in BNF notation):
#
#    <state> := STATE <state_name> <cs_name> [<data>...] [<result>...];
#       <cs_name> := <alpha_num>
#       <state_name> := <alpha_num>
#       <result> := <state_name>
#       <data> := <integer>

#############################################################################
#---------------------------------------------------------------------------
# Initialization
#---------------------------------------------------------------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetTraceLevel   SetTrace            0
stPushFmFiCnt; STATE stPushFmFiCnt     StkPush             0
stInitFmFiCnt; STATE stInitFmFiCnt     StkPopStore         1
stPushLcFiCnt;

STATE stPushLcFiCnt     StkPush             0         stInitLcFiCnt;
STATE stInitLcFiCnt     StkPopStore         2         stDlabSPend;

#---------------------------------------------------------------------------
# Wait for the DLAB bit to be set
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stDlabSPend       CheckTransaction    1         stDlabSNot
                                                      stChkDlabSAddr;
STATE stDlabSNot        ExitStateMachine    0         stDlabSPend;
STATE stChkDlabSAddr    CheckAddress        0x3FB     stDlabSNot
                                                      stChkDlabSBit;
STATE stChkDlabSBit     PushTransData       0         stPushDlabSMask;
STATE stPushDlabSMask   StkPush             0x80      stDlabSAnd;
STATE stDlabSAnd        DataAnd             0         stDlabSPushCmp;
STATE stDlabSPushCmp    StkPush             0x80      stDlabSCompare;
STATE stDlabSCompare    CompareEq           0         stDlabSNot
                                                      stGotoDLsbPend1;
STATE stGotoDLsbPend1   ExitStateMachine    0         stSetDLsbPend1;

#---------------------------------------------------------------------------
# Wait for the Divisor LSB to be set first
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDLsbPend1    CheckTransaction    1         stGotoDLsbPend1
                                                      stChkDivLsbAddr1;
STATE stChkDivLsbAddr1  CheckAddress        0x3F8     stChkDivMsbAddr2
                                                      stGotoDMsbPend1;
STATE stGotoDMsbPend1   ExitStateMachine    0         stSetDMsbPend1;
```

```
#-----------------------------------------------------------------------
# Wait for the Divisor MSB to be set second
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDMsbPend1    CheckTransaction    1         stGotoDMsbPend1
                                                      stChkDivMsbAddr1;
STATE stChkDivMsbAddr1  CheckAddress        0x3F9     stGotoDMsbPend1
                                                      stGotoDlabUPend;
STATE stGotoDlabUPend   ExitStateMachine    0         stDlabUPend;

#-----------------------------------------------------------------------
# Wait for the Divisor MSB to be set first
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stChkDivMsbAddr2  CheckAddress        0x3F9     stGotoDLsbPend1
                                                      stPrtDivSetWarn;
STATE stPrtDivSetWarn   LogWarning          0         stGotoDLsbPend2;
STATE stGotoDLsbPend2   ExitStateMachine    0         stSetDivLsbPend2;

#-----------------------------------------------------------------------
# Wait for the Divisor LSB to be set second
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stSetDivLsbPend2  CheckTransaction    1         stGotoDLsbPend2
                                                      stChkDivLsbAddr2;
STATE stChkDivLsbAddr2  CheckAddress        0x3F8     stGotoDLsbPend2
                                                      stGotoDlabUPend;

#-----------------------------------------------------------------------
# Wait for the DLAB bit to be unset
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#-------------------#---------#-------------------
STATE stDlabUPend       CheckTransaction    1         stDlabUNot
                                                      stChkDlabUAddr;
STATE stDlabUNot        ExitStateMachine    0         stDlabUPend;
STATE stChkDlabUAddr    CheckAddress        0x3FB     stDlabUNot
                                                      stChkDlabUBit;
STATE stChkDlabUBit     PushTransData       0         stPushDlabUMask;
STATE stPushDlabUMask   StkPush             0x80      stDlabUAnd;
STATE stDlabUAnd        DataAnd             0         stPushDlabUCmp;
STATE stPushDlabUCmp    StkPush             0         stDlabUCompare;
STATE stDlabUCompare    CompareEq           0         stDlabUNot
                                                      stGotoSetLcPend;
STATE stGotoSetLcPend   ExitStateMachine    0         stSetLcPend;

#-----------------------------------------------------------------------
# Wait for set of the line control data
#---------------------#-------------------#---------#-------------------
#       State         # Code Segment      # Data    # Trans States
```

```
#--------------------#------------------#---------#------------------
STATE stSetLcPend       CheckTransaction    1         stNotSetLc
                                                      stChkLcAddr;
STATE stNotSetLc        ExitStateMachine    0         stSetLcPend;
STATE stChkLcAddr       CheckAddress        0x3FB     stNotSetLc
                                                      stChkLcFip;

#-------------------------------------------------------------------------
# Fault injection point. Inject a line control data fault at every 'X'
# interval.
#--------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#--------------------#------------------#---------#------------------
STATE stChkLcFip        IncStore            2         stPushLcFiReg;
STATE stPushLcFiReg     StkPushStore        2         stPushLcFiCmp;
STATE stPushLcFiCmp     StkPush             2         stCompLcFi;
STATE stCompLcFi        CompareEq           2         stGotoWritePend
                                                      stSetLcError;
STATE stSetLcError      PushTransData       0         stPushLcErrData;
STATE stPushLcErrData   StkPush             0xFC      stGetLcTransData;
STATE stGetLcTransData  DataAnd             0         stSetLcTransData;
STATE stSetLcTransData  PopTransData        0         stPushLcFi0Cnt;
STATE stPushLcFi0Cnt    StkPush             0         stResetLcFiCnt;
STATE stResetLcFiCnt    StkPopStore         2         stGotoWritePend;

#-------------------------------------------------------------------------
# Wait for the write. If the write is text goto the "wait for write
# completion section. If the write is the DLAB then go back to the
# initalization section.
#--------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#--------------------#------------------#---------#------------------
STATE stWritePend       CheckTransaction    1         stNotWrite
                                                      stChkWriteAddr;
STATE stNotWrite        ExitStateMachine    0         stWritePend;
STATE stChkWriteAddr    CheckAddress        0x3F8     stChkWrtLcAddr
                                                      stGotoWrtVerify;
STATE stChkWrtLcAddr    CheckAddress        0x3FB     stNotWrite
                                                      stWrtChkDlabSBit;
STATE stWrtChkDlabSBit  PushTransData       0         stWrtPshDlabSMask;
STATE stWrtPshDlabSMask StkPush             0x80      stWrtDlabSAnd;
STATE stWrtDlabSAnd     DataAnd             0         stWrtDlabSPushCmp;
STATE stWrtDlabSPushCmp StkPush             0x80      stWrtDlabSCompare;
STATE stWrtDlabSCompare CompareEq           0         stNotWrite
                                                      stGotoDLsbPend1;
STATE stGotoWrtVerify   ExitStateMachine    0         stWrtVerify;

#-------------------------------------------------------------------------
# Wait for the write to complete
#--------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#--------------------#------------------#---------#------------------
STATE stWrtVerify       CheckTransaction    0         stNotWrtVerify
                                                      stChkWrtVfyAddr;
```

```
STATE stNotWrtVerify    ExitStateMachine    0         stWrtVerify;
STATE stChkWrtVfyAddr   CheckAddress        0x3FD     stNotWrtVerify
                                                      stPushLsr;

STATE stPushLsr         PushTransData       0         stPushXmitMask;
STATE stPushXmitMask    StkPush             0x20      stAndXmitStat;
STATE stAndXmitStat     DataAnd             0         stPushXmitCmp;
STATE stPushXmitCmp     StkPush             0x20      stChkXmitStat;
STATE stChkXmitStat     CompareEq           0         stNotWrtVerify
                                                      stChkFmFip;

#-----------------------------------------------------------------------
# Fault injection point. Inject framing errors at every 'X' interval.
#---------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#------------------#---------#------------------
STATE stChkFmFip        IncStore            1         stPushFmFiReg;
STATE stPushFmFiReg     StkPushStore        1         stPushFmFiCmp;
STATE stPushFmFiCmp     StkPush             1000      stCompFmFi;
STATE stCompFmFi        CompareEq           1000      stChkWrtError
                                                      stSetFmError;
STATE stSetFmError      PushTransData       0         stPushFmErrData;
STATE stPushFmErrData   StkPush             0x08      stGetFmTransData;
STATE stGetFmTransData  DataOr              0         stSetFmTransData;
STATE stSetFmTransData  PopTransData        0         stPushFmFi0Cnt;
STATE stPushFmFi0Cnt    StkPush             0         stResetFmFiCnt;
STATE stResetFmFiCnt    StkPopStore         1         stGotoResetPend;

#-----------------------------------------------------------------------
# Check for errors on the write
#---------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#------------------#---------#------------------
STATE stChkWrtError     PushTransData       0         stPushWrtFmMask;
STATE stPushWrtFmMask   StkPush             0x08      stWrtStatFmAnd;
STATE stWrtStatFmAnd    DataAnd             0         stPushCmpFm;
STATE stPushCmpFm       StkPush             0x08      stCmpFmError;
STATE stCmpFmError      CompareEq           0         stGotoWritePend
                                                      stGotoResetPend;
STATE stGotoWritePend   ExitStateMachine    0         stWritePend;
STATE stGotoResetPend   ExitStateMachine    0         stResetPend;

#-----------------------------------------------------------------------
# Wait for the reset
#---------------------#------------------#---------#------------------
#       State         # Code Segment      # Data    # Trans States
#---------------------#------------------#---------#------------------
STATE stResetPend       CheckTransaction    1         stNotResetWrite
                                                      stChkResetAddr;
STATE stNotResetWrite   ExitStateMachine    0         stResetPend;
STATE stChkResetAddr    CheckAddress        0x3FD     stLogResetErr
                                                      stGotoWritePend;
# Check what is set?
STATE stLogResetErr     LogViolation        0         stChkWriteAddr;
```

##

8 Appendix B - Sample State Machine Implementation

```
 /*--------------------------------------------------------------------
 * FILE NAME: state_machine.cpp
 *
 * IMPORTANT:  READ BEFORE DOWNLOADING, COPYING, INSTALLING OR USING.
 * By downloading, copying, installing or using the software you agree
 * to this license.  If you do not agree to this license, do not
 * download, install, copy or use the software.
 *
 *                          Intel Open Source License
 *
 * Copyright (c) 2002 Intel Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * #  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * #  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * #  Neither the name of the Intel Corporation nor the names of its
 *    contributors may be used to endorse or promote products derived from
 *    this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE AND NONINFRINGEMENT ARE DISCLAIMED. IN NO EVENT SHALL INTEL OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *--------------------------------------------------------------------*\

/*--------------------------------------------------------------------
 *    DESCRIPTION:  This program is and example program of how one could
 *                implement a finite state machine.  This has been written
 *                as and example to be used in a class.  The doucmentation
```

```
 *               in the source is limited.  It is also assumed that the
 *               person being taught has a general understanding of
 *               what a state machine is.
 *
 *    Agruments:    [{TRACE [=] {ON|OFF}|?}]
 *
 *                   TRACE is used to turn trace of the statemachine on
 *                   or off.  The default is off.  This will stay set
 *                   for the complete run of the program.  Currenlty no
 *                   method is coded to allow trace to be controlled during
 *                   runtime.
 *
 *                   This command is not case sensitive.
 *
 *    AUTHOR:   Donald W. Long
 *
 *    HISTORY:  1.0     -   First Release
 *-----------------------------------------------------------------------*/

#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

    // General defines
#define ProgramVersion          "1.0"
#define ProgramVersionDate      "10-3-2001"
#define TRUE                    -1
#define FALSE                   0

// These are the code seqment names with the values that they can return
// All sets of code segment return values must start with 0 and go
// up by 1, no holes allowed.
typedef enum {
    CS_PrintBanner = 0,
    CS_AskForNum1,
    CS_AskForNum2,
    CS_AskForFunc,
    CS_Add,
    CS_Sub,
    CS_Times,
    CS_Divide,
    CS_OutPutResults,
    CS_Exit,
    CS_BadInput
} CodeSeqmentTF;

   // Misc Defines
#define CS_PrintBanner_OK       0

#define CS_AskForNum1_OK        0
#define CS_AskForNum1_BAD       1
#define CS_AskForNum1_Exit      2
```

```
#define CS_AskForNum2_OK        0
#define CS_AskForNum2_BAD       1
#define CS_AskForNum2_Exit      2

#define CS_AskForFunc_Add       0
#define CS_AskForFunc_Sub       1
#define CS_AskForFunc_Times     2
#define CS_AskForFunc_Divide    3
#define CS_AskForFunc_Exit      4
#define CS_AskForFunc_Unknown   5

#define CS_Add_OK               0

#define CS_Sub_OK               0

#define CS_Times_OK             0

#define CS_Divide_OK            0

#define CS_OutPutResults_OK     0

#define CS_Exit_OK              0

#define CS_BadInput_OK          0

// This table is used to output the text name of the code segement
// and its values if trace is turned on.
// Format:  <Num Values>  <code segment name>  <Names of values,
//                                              occurs for <Num Values>>
// The Value string names are in the order they are defined
//    (i.e., 0, 1, 2, ...)
//
// At the end of each code segment value list a -1 must occur.
// At the end of the table another -1 must occur.
//
static char CodeSeqmentNames[] =    {
    1, 'P', 'r', 'i', 'n', 't', 'B', 'a', 'n', 'n', 'e', 'r', 0,
        'P', 'r', 'i', 'n', 't', 'B', 'a', 'n', 'n', 'e', 'r',
        '_', 'O', 'K', 0,
    -1,
    3, 'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'O', 'K', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'B', 'A', 'D', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '1',
             '_', 'E', 'x', 'i', 't', 0,
    -1,
    3, 'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
             '_', 'O', 'K', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
             '_', 'B', 'A', 'D', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'N', 'u', 'm', '2',
```

```
            '_', 'E', 'x', 'i', 't', 0,
    -1,
    6, 'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'A', 'd', 'd', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'S', 'u', 'b', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'T', 'i', 'm', 'e', 's', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'D', 'i', 'v', 'i', 'd', 'e', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'E', 'x', 'i', 't', 0,
        'A', 's', 'k', 'F', 'o', 'r', 'F', 'u', 'n', 'c',
             '_', 'U', 'n', 'k', 'n', 'o', 'w', 'n', 0,
    -1,
    1, 'A', 'd', 'd', 0,
        'A', 'd', 'd', '_', 'O', 'K', 0,
     -1,
    1, 'S', 'u', 'b', 0,
        'S', 'u', 'b', '_', 'O', 'K', 0,
    -1,
    1, 'T', 'i', 'm', 'e', 's', 0,
        'T', 'i', 'm', 'e', 's', '_', 'O', 'K', 0,
    -1,
    1, 'D', 'i', 'v', 'i', 'd', 'e', 0,
        'D', 'i', 'v', 'i', 'd', 'e', '_', 'O', 'K', 0,
    -1,
    1, 'O', 'u', 't', 'P', 'u', 't', 'R', 'e', 's', 'u',
             'l', 't', 's', 0,
        'O', 'u', 't', 'P', 'u', 't', 'R', 'e', 's', 'u',
             'l', 't', 's', '_', 'O', 'K', 0,
    -1,
    1, 'E', 'x', 'i', 't', 0,
        'E', 'x', 'i', 't', '_', 'O', 'K', 0,
    -1,
    1, 'B', 'a', 'd', 'I', 'n', 'p', 'u', 't', 0,
        'B', 'a', 'd', 'I', 'n', 'p', 'u', 't', '_', 'O', 'K', 0,
    -1,
    -1
};

// This gives us the names of the states we will be using.
// It should be noted that this order must also be followed
// in the StateTable.
typedef enum {
    ST_Start        = 0,
    ST_GetNum1      = 1,
    ST_GetNum2      = 2,
    ST_GetFunc      = 3,
    ST_Add          = 4,
    ST_Sub          = 5,
    ST_Times        = 6,
    ST_Divide       = 7,
```

```
    ST_OutPutR      = 8,
    ST_Exit         = 9,
    ST_Num1Bad      = 10,
    ST_Num2Bad      = 11,
    ST_FuncBad      = 12
} States_TF;


int *StateTablePtr = 0; // This is the location in the state table
                        // we are at and the state table.
#define StateTableNumElements   8   // If you add the ability to have more
                                    // values you must increase this
                                    // by that number.

// Format:  <State> <Code Segment>  <state for val1> <state for val2>
//                                  <state for val3> <state for val4>
//                                  <state for val5> <state for val6>
//
//   If the value is not used then set to -1.
//
//   The order of the states must match the order of the states
//   defined in StateTF.
static int  StateTable[] =  {
    ST_Start,   CS_PrintBanner,  ST_GetNum1,     -1,         -1,         -1,
             -1,         -1,
    ST_GetNum1, CS_AskForNum1,   ST_GetNum2,     ST_Num1Bad, ST_Exit,    -1,
             -1,         -1,
    ST_GetNum2, CS_AskForNum2,   ST_GetFunc,     ST_Num2Bad, ST_Exit,    -1,
             -1,         -1,
    ST_GetFunc, CS_AskForFunc,   ST_Add,         ST_Sub,     ST_Times,   ST_Divide,
            ST_Exit,    ST_FuncBad,
    ST_Add,     CS_Add,          ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Sub,     CS_Sub,          ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Times,   CS_Times,        ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_Divide,  CS_Divide,       ST_OutPutR,     -1,         -1,         -1,
             -1,         -1,
    ST_OutPutR, CS_OutPutResults, ST_GetNum1,    -1,         -1,         -1,
             -1,         -1,
    ST_Exit,    CS_Exit,         ST_Exit,        -1,         -1,         -1,
             -1,         -1,
    ST_Num1Bad, CS_BadInput,     ST_GetNum1,     -1,         -1,         -1,
             -1,         -1,
    ST_Num2Bad, CS_BadInput,     ST_GetNum2,     -1,         -1,         -1,
             -1,         -1,
    ST_FuncBad, CS_BadInput,     ST_GetFunc,     -1,         -1,         -1,
             -1,         -1
};

    // State names, format is <name><null> .... Order must match the order
    // of the state values (see StateTF).
    // Last byte is -1 to show end of table.
```

```
static char StateNames[] =  {
    'S', 't', 'a', 'r', 't', 0,
    'G', 'e', 't', 'N', 'u', 'm', '1', 0,
    'G', 'e', 't', 'N', 'u', 'm', '2', 0,
    'G', 'e', 't', 'F', 'u', 'n', 'c', 0,
    'A', 'd', 'd', 0,
    'S', 'u', 'b', 0,
    'T', 'i', 'm', 'e', 's', 0,
    'D', 'i', 'v', 'i', 'd', 'e', 0,
    'O', 'u', 't', 'P', 'u', 't', 'R', 0,
    'E', 'x', 'i', 't', 0,
    'N', 'u', 'm', '1', 'B', 'a', 'd', 0,
    'N', 'u', 'm', '2', 'B', 'a', 'd', 0,
    'F', 'u', 'n', 'c', 'B', 'a', 'd', 0,
    -1
};

    // General flags for the program
int Trace = FALSE;      // if not 0 then we will trace the statemachine

    // Function prototypes
int Init(int argc, char *argv[]);
int PrintTrace( int  *StateTableEntry, char *CodeSeqmentNames,
                char *StateNames,      int CodeSegmentValue);
int ConvertToNum(char *InputData, double *Result);
char *StripSpaces(char *Line);
void ToUpper(char *Data);

/*
 ** main
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     argc    -   number of argments passed to the program
 *                  argv    -   address list of arguments.
 *
 *  DESCRIPTION:    main entry point for program
 *  RETURNS:        0   -   Program terminates OK
 *                  1   -   Program had an internal error
 *                  2   -   Invalid argument passed to program
 */
int main(int argc, char* argv[])
{
  int     CodeSegmentValue=-1;  // Code segment value returned from
                                // switch statement
  double  Num1=0;               // Contains the last value inputed
                                // for CS_AskForNum1
  double  Num2=0;               // Contains the last value inputed
                                // for CS_AskForNum2
  double  FuncResult=0;         // Results of the operation requested.
  char    InputData[100];       // String to store the input data into
                                // for all cin operations.
  char    *ptr;                 // General pointer.
```

```
    // Init the program
if (!Init(argc, argv)) {
    cerr << endl <<  "**** Invalid arguments passed ****"  << endl <<
            "SampleStateMachine [{TRACE [=] {ON|OFF}|?}]"   << endl << endl;
    return(2);
}
    // Main loop for program
StateTablePtr=StateTable;
while (TRUE) {
    switch (StateTablePtr[1]) {
        case CS_PrintBanner:
            cout << "Test StateMachine (" << ProgramVersion << "-" <<
                                ProgramVersionDate << ") ** TRACE = ";
            if (Trace==TRUE) {
                cout << "ON";
            } else {
                cout << "OFF";
            }
            cout << endl << endl;
            CodeSegmentValue=CS_PrintBanner_OK;
            break;
        case CS_AskForNum1:
            cout << endl << "Please enter the first number or 'exit'? ";
            cin >> InputData;

            ptr = StripSpaces(InputData);
            ToUpper(ptr);

                // Process the line
            if (!strcmp(ptr, "EXIT")) {
                CodeSegmentValue=CS_AskForNum1_Exit;
            } else {
                if (ConvertToNum(ptr, &Num1)) {
                    CodeSegmentValue=CS_AskForNum1_OK;
                } else {
                    CodeSegmentValue=CS_AskForNum1_BAD;
                }
            }
            break;
        case CS_AskForNum2:
            cout << endl << "Please enter the second number or 'exit'? ";
            cin >> InputData;

            ptr = StripSpaces(InputData);
            ToUpper(ptr);

                // Process the line
            if (!strcmp(ptr, "EXIT")) {
                CodeSegmentValue=CS_AskForNum2_Exit;
            } else {
                if (ConvertToNum(ptr, &Num2)) {
                    CodeSegmentValue=CS_AskForNum2_OK;
                } else {
                    CodeSegmentValue=CS_AskForNum2_BAD;
```

```
        }
    }
    break;
case CS_AskForFunc:
    cout << endl << "Please enter the function to perform"  << endl <<
                    "  Add"                          << endl <<
                    "  Sub[tract]"                   << endl <<
                    "  Times"                        << endl <<
                    "  Div[ide]"                     << endl <<
                    "  Exit"                         << endl <<
                    "? ";
    cin >> InputData;
    ptr = StripSpaces(InputData);
    ToUpper(ptr);
    if (!strcmp(ptr, "ADD")) {
        CodeSegmentValue=CS_AskForFunc_Add;
    } else if (!strcmp(ptr, "SUB") || !strcmp(ptr, "SUBTRACT")) {
        CodeSegmentValue=CS_AskForFunc_Sub;
    } else if (!strcmp(ptr, "TIMES")) {
        CodeSegmentValue=CS_AskForFunc_Times;
    } else if (!strcmp(ptr, "DIV") || !strcmp(ptr, "DIVIDE")) {
        CodeSegmentValue=CS_AskForFunc_Divide;
    } else if (!strcmp(ptr, "EXIT")) {
        CodeSegmentValue=CS_AskForFunc_Exit;
    } else {
        CodeSegmentValue=CS_AskForFunc_Unknown;
    }
    break;
case CS_Add:
    FuncResult=Num1+Num2;
    CodeSegmentValue=CS_Add_OK;
    break;
case CS_Sub:
    FuncResult=Num1-Num2;
    CodeSegmentValue=CS_Sub_OK;
    break;
case CS_Times:
    FuncResult=Num1*Num2;
    CodeSegmentValue=CS_Times_OK;
    break;
case CS_Divide:
    FuncResult=Num1/Num2;
    CodeSegmentValue=CS_Divide_OK;
    break;
case CS_OutPutResults:
    cout << endl << "Your results are " << FuncResult << endl << endl;
    CodeSegmentValue=CS_OutPutResults_OK;
    break;
case CS_Exit:
    CodeSegmentValue=CS_Exit_OK;
    return(0);
case CS_BadInput:
    CodeSegmentValue=CS_BadInput_OK;
    cout << "Input is not valid - " << InputData << endl;
```

```
            break;
        default:
        cerr << endl <<
        "*********************************************************" << endl <<
        "*********************************************************" << endl <<
        "**                    INTERNAL ERROR                   **" << endl <<
        "**          The Code Segment Does not exist            **" << endl <<
        "** Either state table is bad, code segment not added,  **" << endl <<
        "**     or logic error in moving thru state table       **" << endl <<
        "*********************************************************" << endl <<
        "*********************************************************" << endl
        << endl;
          return(1);
    }
    if (Trace==TRUE) {
        if (!PrintTrace(StateTablePtr, CodeSeqmentNames,
            StateNames, CodeSegmentValue)) {
            return(1);
        }
    }
    StateTablePtr=&(StateTable[(StateTablePtr[CodeSegmentValue+2])
                    * StateTableNumElements]);
  }
}

/*
 ** Init
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     argc    -   number argument passed
 *                  argv    -   Address array to arguments
 *
 *  DESCRIPTION:    This will parse out the program arguments.  If
 *                  any errors will exit with a value of 2.
 *
 *  RETURNS:        0   -   Agruments invalid
 *                  1   -   Agruments processed
 *
 */
int Init(int argc, char *argv[])
{
    char *mode;
    char *ptr;


    if (argc>1) {
        if (argc<5 && argc >2) {
            if (argc==4) {
                mode=argv[3];
                ptr=StripSpaces(argv[2]);
                if (strcmp(ptr, "=")) {
                    return(0);
                }
```

```
            } else {
                mode=argv[2];
            }
            ptr=StripSpaces(argv[1]);
            ToUpper(ptr);
            if (!strcmp(ptr, "TRACE")) {
                mode=StripSpaces(mode);
                ToUpper(mode);
                if (!strcmp(mode, "ON")) {
                    Trace=TRUE;
                } else if (!strcmp(mode, "OFF")) {
                    Trace=FALSE;
                } else {
                    return(0);
                }
            } else {
                return(0);
            }
        } else {
            if (argc==2) {
                if (!strcmp(argv[1], "?")) {
                    cout << endl << "SampleStateMachine [{TRACE [=] {ON|OFF}|?}]"
                                                          << endl << endl;
                    exit(0);
                } else {
                    return(0);
                }
            } else {
                return(0);
            }
        }
    }
    return(-1);
}

 /*
 ** PrintTrace
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:
 *    StateTableEntry  - Pointer to the current state table location
 *                       that is being processed
 *    CodeSegmentNames - Pointer to the table that contains all the
 *                       code segment names and they code segment values
 *    StateNames       - Pointer to the table that contains the State Names.
 *    CodeSegmentValue - The value that was returned from the last code
 *                       segment that was executed for the current state.
 *
 *  DESCRIPTION:
 *    This will take the current state information (after execution) and
 *    output in text what has occured and the new state that will occur.
 *
 *    The output goes to cerr and is in the following format
```

```
 *       <CurState>(val) - <CodeSegment>(val) - <Codesegment value>(val) ->
 *                                                    <NewState>(val)<eol>
 *  RETURNS:
 */
int PrintTrace( int  *StateTableEntry, char *CodeSeqmentNames,
                char *StateNames,      int CodeSegmentValue)
{
    int     CurState;
    int     NewState;
    int     CodeSegment;
    char    *CurStateName;
    char    *NewStateName;
    char    *CodeSegmentName;
    char    *CodeSegmentValName;
    int     NumCodeSegments;
    int     i;
    int     ii;


        // Get the items from the state table entry.
    CurState    = StateTableEntry[0];
    CodeSegment = StateTableEntry[1];
    NewState    = StateTableEntry[CodeSegmentValue+2];

        // Get the current state name.
    for (CurStateName=StateNames, i = 0;
      i<CurState && CurStateName[0]!=-1; i++) {
        for (; *CurStateName!=0; CurStateName++);
        CurStateName++;
    }
    if (*CurStateName==-1) {
        cerr << endl <<
         "*******************************************" << endl <<
         "*******************************************" << endl <<
         "**               INTERNAL ERROR            **" << endl <<
         "**  Current State Not In State Name Table  **" << endl <<
         "** State table is  bad or State Name Table **" << endl <<
         "*******************************************" << endl <<
         "*******************************************" << endl << endl;
        return(0);
    }

        // Get the new state name.
    for (NewStateName=StateNames, i = 0;
      i<NewState && NewStateName[0]!=-1; i++) {
        for (; *NewStateName!=0; NewStateName++);
        NewStateName++;
    }
    if (*NewStateName==-1) {
        cerr << endl <<
         "*******************************************" << endl <<
         "*******************************************" << endl <<
         "**               INTERNAL ERROR            **" << endl <<
         "**      New State Not In State Name Table  **" << endl <<
```

```
     "** State table is  bad or State Name Table **" << endl <<
     "*****************************************" << endl <<
     "*****************************************" << endl << endl;
    return(0);
}

    // Get the codesegment name.
for (CodeSegmentName=CodeSeqmentNames, i=0;
  i<CodeSegment && CodeSegmentName[0]!=-1; i++) {
    NumCodeSegments=CodeSegmentName[0];
    for (; *CodeSegmentName!=0; CodeSegmentName++);
    CodeSegmentName++;

        // Skip the codesegment return value names.
    for (ii=0; ii<NumCodeSegments && CodeSegmentName[0]!=-1; ii++) {
        for (; *CodeSegmentName!=0; CodeSegmentName++);
        CodeSegmentName++;
    }
    if (ii!=NumCodeSegments && CodeSegmentName[0]!=-1) {
        cerr << endl <<
      "*************************************************" << endl <<
      "*************************************************" << endl <<
      "**            INTERNAL ERROR                   **" << endl <<
      "** Code Segment Vak Not in Code Segment Table **" << endl <<
      "** Code Segment is bad  or Code Segment Table **" << endl <<
      "*************************************************" << endl <<
      "*************************************************" << endl << endl;
        return(0);
    }
    CodeSegmentName++;  // Skip the -1 at the end of the
                        // codesegment values.
}
if (*CodeSegmentName==-1) {
    cerr << endl <<
     "*************************************************" << endl <<
      "*************************************************" << endl <<
      "**              INTERNAL ERROR                 **" << endl <<
      "**   Code Segment Not in Code Segment Table  **" << endl <<
      "** Code Segment is bad or Code Segment Table **" << endl <<
      "*************************************************" << endl <<
      "*************************************************" << endl << endl;
    return(0);
}

    // Setup for getting the codesegment value name.
NumCodeSegments=CodeSegmentName[0];
CodeSegmentName++;
for (CodeSegmentValName=CodeSegmentName; *CodeSegmentValName!=0;
  CodeSegmentValName++);
CodeSegmentValName++;

    // Get the code segment value name
for (i=0; i<CodeSegmentValue && CodeSegmentValName[0]!=-1; i++) {
    for (; *CodeSegmentValName!=0; CodeSegmentValName++);
```

```
        CodeSegmentValName++;
    }
    if (CodeSegmentName[0]==-1) {
        cerr << endl <<
          "***********************************************" << endl <<
          "***********************************************" << endl <<
          "**            INTERNAL ERROR                 **" << endl <<
          "** Code Segment Vak Not in Code Segment Table **" << endl <<
          "** Code Segment is bad  or Code Segment Table **" << endl <<
          "***********************************************" << endl <<
          "***********************************************" << endl << endl;
        return(0);
    }

        // Output format is <CurState>(val) - <CodeSegment>(val) -
                            <Codesegment value>(val) -> <NewState>(val)
    cerr << CurStateName    << "(" << CurState          << ") - "   <<
         CodeSegmentName    << "(" << CodeSegment       << ") - "   <<
         CodeSegmentValName << "(" << CodeSegmentValue  << ") -> "  <<
         NewStateName       << "(" << NewState          << ")"      << endl;
    return(-1);
}

/*
 ** ConvertToNum
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:  InputData   -   String that is to be converted to a number
 *               Result      -   Address to a double to put the results in
 *
 *  DESCRIPTION: This function will take an ascii string and convert it to
 *               a double.  This function assumes all spaces have been
 *               removed from the start of the string and the end.
 *
 *  RETURNS:     -1  -   Converted ok
 *                0  -   Invalid data in InputData (not a number)
 *
 */
int ConvertToNum(char *InputData, double *Result)
{
    char *ptr;


        // Make sure all charactors are valid.
    for (ptr=InputData; *ptr!=0; ptr++) {
        if (!isdigit(*ptr)) {
            if (*ptr=='.') {    // Floating Point Number.
                for (ptr++; *ptr!=0; ptr++) {
                    if (!isdigit(*ptr)) {
                        return(0);
                    }
                }
                break;  // Leave the for loop so we can do the atof function.
```

```
            } else {
                return(0);
            }
        }
    }
    *Result=atof(InputData);
    return(-1);
}

/*
 ** StripSpaces
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     Line    -   The line to remove spaces
 *
 *  DESCRIPTION:    This will remove all the spaces at the start and end
 *                  of Line.
 *
 *  RETURNS:        Address of first charactor in Line that is not a space
 *
 */
char *StripSpaces(char *Line)
{
    char *stptr;
    char *ptr;


        // Strip off all spaces
    for (stptr=Line; *stptr == ' ' && *stptr != 0; stptr++);
    for (ptr=stptr; *ptr != ' ' && *ptr != 0; ptr++);
    *ptr=0;
    return(stptr);
}

/*
 ** ToUpper
 *
 *  FILENAME: D:\Projects\SampleStateMachine\samplestatemachine.cpp
 *
 *  PARAMETERS:     Data    -   Data to convert to upper case
 *
 *  DESCRIPTION:    This function will convert a staring to upper case
 *
 *  RETURNS:
 *
 */
void ToUpper(char *Data)
{
    char *ptr;

    for (ptr=Data; *ptr!=0; ptr++) {
        *ptr=toupper(*ptr);
    }
```

```
}
```