

Testable Embedded System Firmware Development: The Out-In Methodology

Narayanan (Nary) Subramanian
Firmware Engineer
Anritsu Company
Richardson, TX 75081

Lawrence Chung
Dept. of Computer Science
University of Texas, Dallas
Richardson, TX 75081

Abstract.

Reliability is of paramount importance to just about any embedded system firmware. This paper presents the out-in methodology, a new reliability-driven approach to developing such a system, which is intended to detect static and, more importantly, dynamic errors much faster than the usual firmware development methods. In this approach the core functionality is developed together with an interface software that is used specifically for testing the core functionality. This paper describes the use of this approach in a real life situation and discusses the benefits, potential pitfalls, and possible other application areas.

1. INTRODUCTION

Embedded systems are widely prevalent in the modern world. They include household items such as microwaves, dishwashers, telephones; portable systems like cell-phones, handheld personal computers, robots; and covering the spectrum at the higher end of complexity are the space ships and space shuttles. All of these systems have a dedicated hardware that includes at least one CPU running dedicated software. This paper is concerned with the development of error-free software for such embedded systems. (In this paper, “embedded system” and “system” are used synonymously.)

Software for an embedded system has two basic components: Control and I/O [2]. The control component is the heart of the system and does the processing and other tasks that distinguish the particular system in which it resides from the others. The I/O component handles the processing associated with inputs and outputs – it includes inputs from the user, the environment and any other source of relevance to the system, and outputs that the system sends to associated peripherals. In a microwave oven, for example, the keypad is the input source, the LCD display is the output, and the control reacts to the inputs and displays output, turns the oven off and on and performs the functions of the microwave oven.

An important characteristic required of such embedded systems is reliability. When the user gives a command to the embedded system to do some activity, the system should either do what it is told to or else display an error message in case the command was wrong in the current context. However, if the software has not been designed with care, there could be some sequence of user-system interactions that lead to a system crash. This usually manifests itself by making the system completely unresponsive to any further user interaction. The only resort for the user is then to reboot the system. Such an occurrence, as can be expected, results in poor user confidence in the system.

Such user-interaction problems can be overcome by careful scenario analysis upfront, followed by testing after the implementation of the software. While for simple embedded systems, this approach may be sufficient to produce an error-free working system, for complex systems that include multiple tasks and multiple processors, such an approach is far from sufficient to ensure error-free software. This is because such systems have complicated hardware-software interactions that are difficult to visualize before the entire system (hardware and software) is built. Hence it is difficult for the development teams of such systems to plan for such problems. Detection of such problems requires non-invasive testing of the system while the system is running. Automatic testing (or machine-based testing) is required since tests may have to be repeated many, many times (thousands to hundreds of thousands of times) to wring out some of these hardware-software interaction problems. However, most currently used testing methods cannot do such tests.

The out-in methodology that this paper proposes aims to ensure the following:

1. that the software is developed in a way that makes non-invasive testing at run-time feasible
2. there is an automatic method to test the software at run-time.

The subsequent sections discuss this new methodology in detail and clarify the above points with numerous examples.

The popular IEEE488 (also known as GPIB or General Purpose Interface Bus) was used for implementing OIM in this paper. This is a parallel bus and has a dedicated service request line that can be used to indicate completion of tasks. The reader is not required to know the details of this bus. Any data specific to this bus is explained in the paper wherever necessary. Also since the implementation was done for a test and measuring instrument, which is itself an embedded system, the word “instrument” has been used synonymously with the words “embedded system”. One more point – in the discussion that follows there will be statements like “upon completion of measurement ...”. This means that the instrument completed the measurement it was asked to do and raised the service request line of the IEEE488 bus. This lets the PC know when the measurement was completed and take further action.

Section 2 discusses the current firmware development process and points out its drawbacks. Section 3 discusses the out-in methodology in detail. Section 4 discusses an implementation, while Section 5 discusses the use of the implementation of this new methodology. Section 6 draws conclusions from this work.

2. THE CURRENT FIRMWARE DEVELOPMENT PROCESS

Firmware is usually developed using either the classical waterfall or the incremental model of development (for a discussion of current firmware development methodology please see [4] and [5]). Once the firmware has been developed, testing is done during the verification stage of the development cycle. Usually black-box testing [6] is done at this stage. Regression tests are also performed. These tests may be manual or automatic. Since automatic testing is faster, many of these tests are automated. However, most of the tests cannot test the firmware *in situ*, i.e., as the firmware is running (see [7] for description of automated testing). For example, if boundary value testing is to be done, then an automated test will test the software for the extreme values – but it does not test how the software will behave if these boundary values are given when the system is running. Of course, the system can be manually tested by actually entering the boundary values (for a system that lets users set values of data) and checking to see if the system behaves as expected. But this is very time consuming. Another alternative is to have the system test itself upon start-up or upon pressing a special key – but this will require a pre-defined sequence of tests and will be extremely inflexible. If a test fails there is no easy way of identifying why it failed.

In fact, the following drawbacks in the current firmware development methodology can be observed:

1. No single “window” that can access all parts of the system.
2. There is no facility for obtaining data from the firmware at run time – no way to confirm correctness at run time automatically (“printf”s” inserted in the code do not allow for automated interpretation).
3. Tests are automated but cannot test the firmware while it is running (automated testing is done on passive code, not on the run-time code; for an embedded system its run-time behavior is what is observed by the customer and hence there is an urgent need for automatic run-time tests).

The out-in methodology that is presented in the next section aims to overcome these problems.

3. THE OUT-IN METHODOLOGY

The first requirement for testing a system *in situ* is to have some means for extracting data out of the running system. One way of doing this for a system with a display is to have print statements inserted at strategic locations in the code that would then send the data to the display. The disadvantage with this method is that the observation of the outputs can be done only manually. Another way of doing this is to have some means of reading out the data such as through a network interface. This is the technique that the out-in methodology (OIM) uses. The OIM approaches the problem by intentionally developing the network interface to the firmware, even though the system may not need such an interface. In fact the first step in the firmware development process using OIM is to develop a computer interface. The only reason that this interface is developed is for testing – all sorts of tests can be performed with such an interface as will be explained later. The OIM system configuration is given in Figure 1.

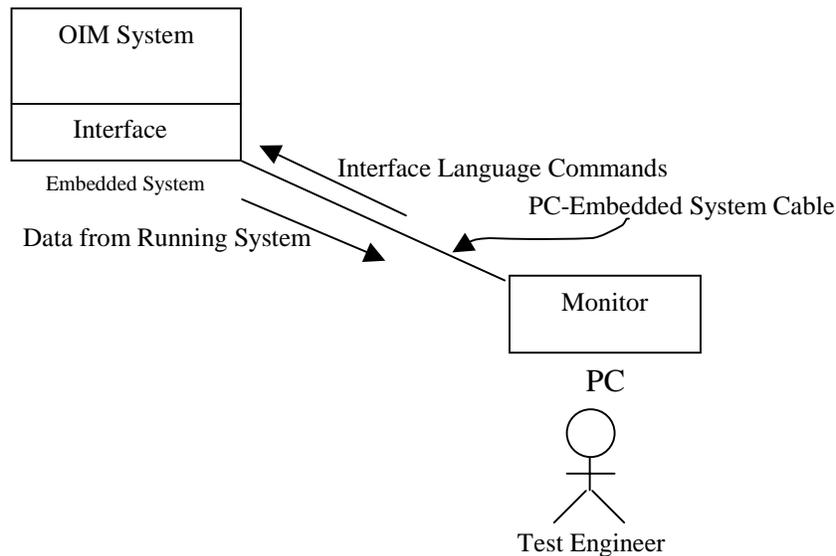


Figure 1: OIM System Configuration

As can be seen from Figure 1, the embedded system uses an external PC for testing. The PC is connected to the OIM system through a hardware interface. The test engineer runs the tests from the PC. The test suite is called the monitor. The embedded system receives the commands (called the Interface Language Commands) from the PC over the PC-Embedded System cable and executes the commands. Any responses that the OIM system has to send to the PC (because of the commands received) are also sent over the PC-Embedded System cable. The OIM requirements added the external PC interface and the monitor on the PC. The difference from automated tests is that the PC now takes data out of running system (or the executing code) while the normal usage of automated testing refers to automatically testing the passive, static code.

The difference between traditional firmware development process and the OIM is depicted in Figure 2. As can be seen in the figure, computer interface (shown as computer I/O) is an optional firmware item for traditional process – if the requirements call for such an interface it is included, else it is excluded. In OIM, irrespective of the requirements, a computer I/O is the first firmware item that is developed. All other software is developed later. What is the advantage of this approach? The computer interface is the window to all other parts of the software. In fact, by proper design (and this is not difficult) it can be ensured that no part of the software is inaccessible from the interface. How is this done? This is accomplished by having an interface language command for each item of the software that has to be accessed. Please note that this command set is developed only for testing purposes. It is quite possible that the computer interface was a

legal requirement and the instrument was required to support a standard set of commands. However, the commands that the OIM requires the instrument to support are in addition to the standard ones. These additional commands are used only to exercise the system. The OIM development process is described in detail in the next section.

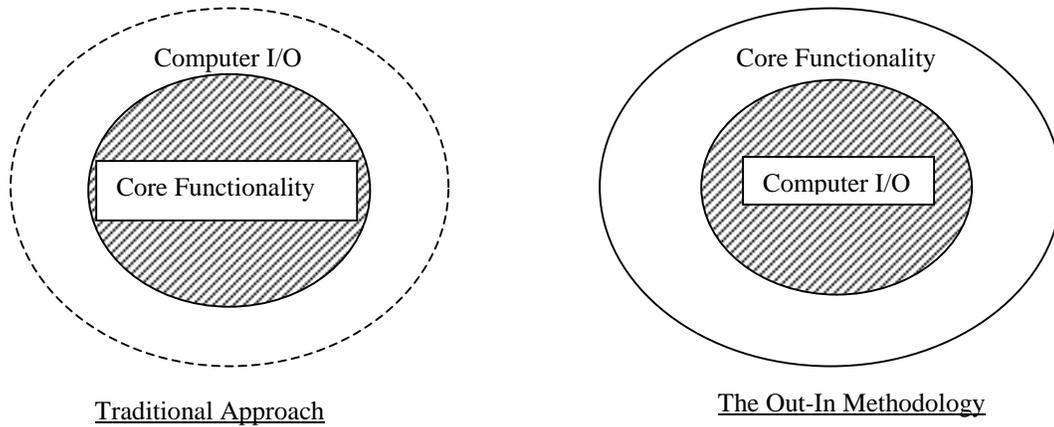


Figure 2: Out-In vs. Traditional Methodology Comparison

The testing method described in this paper has been described elsewhere ([10], [11]). However, these methods do not affect the firmware development process of the embedded system they are connected to in any way. In the authors' view this is a huge loss of opportunity to test the firmware as well. The OIM methodology incorporates the feasibility for external PC testing in the firmware development process and thus exploits the advantages that a PC-based testing offers.

3.1 OIM Firmware Development Process

The firmware development process for OIM is shown in Figure 3. In order to explain this process the following example, which is a part of the actual requirements, is used:

The instrument shall perform test A in which it shall measure three parameters V1, V2, V3 (which are physical parameters) and compute the value of parameter V4 by the formula F:

$$V4 = F(V1, V2, V3).$$

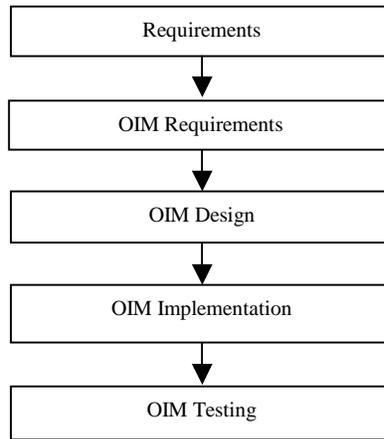


Figure 3: OIM Firmware Development Process

As can be seen from Figure 3, the OIM firmware development process differs from the usual software development process after the requirements stage. After the initial requirements stage the subsequent phases of development are OIM Requirements phase, OIM Design phase, OIM Implementation phase and finally the OIM Testing phase. The OIM firmware development process is explained using the example in Figure 4. As can be seen from this figure, the first step in the OIM methodology is the requirements step. Once the requirements are collected, the next step is to enhance the requirements by adding the OIM specific requirements. There are three parts in an OIM system – the Computer Interface, the Instrument and the Monitor. The requirements stage chooses the computer interface to be used for OIM, defines the requirements for the Instrument and the Monitor. The OIM Design phase designs the Interface Language Commands that will be used over the computer interface, the Instrument firmware and the Monitor tests. The OIM Implementation phase implements the three parts. The OIM Testing phase tests the Monitor first using standard PC-based tests while the Instrument firmware is tested by the Monitor using the Interface Language Commands. This also tests the Interface Language Commands itself. These phases are explained in detail later. The state transition diagrams for the Instrument and the PC are given in Figure 4a, and the sequence diagram is given in Figure 4b.

State Transition Diagrams. As shown in Figure 4a, the instrument is initially in the Idle state. The moment it receives START A command from the monitor, the instrument goes to Perform Test A state. Once the instrument completes the test it indicates completion of the test (in GPIB it does so by raising the service request line of the IEEE488 interface) and returns to the Idle state. Whenever the instrument receives any one of GET? V1, GET? V2, GET? V3 or GET? V4 from the monitor, the instrument goes to Read Results state and sends the values of V1, V2, V3 and V4, respectively, to the monitor. After sending the response to the monitor the instrument returns to the Idle state and awaits further interaction from the monitor.

The monitor is initially in the Idle state. As soon as the test engineer starts the test for Test A, the monitor sends the command START A to the instrument and waits for the instrument to complete Test A. The monitor then sends out commands GET? V1, GET? V2 and GET? V3 to the instrument and reads the values of V1, V2 and V3. The monitor then computes its value of V4, say V4', using the formula F. The monitor then reads V4 calculated by the instrument by sending the command GET? V4 to the instrument and compares V4 with V4'. If they are different the monitor raises an alarm and informs the test engineer about the error. Else the monitor ends the test or may inform the test engineer of the successful completion of the test.

Sequence Diagram. The sequence diagram is shown in Figure 4b. The test engineer first starts Test A. This causes the monitor in the PC to send the command START A to the instrument. Upon receiving this command, the instrument performs Test A the completion of which is then indicated to the monitor. The monitor then sends the commands GET? V1, GET? V2 and GET? V3 one after the other and for each

command receives the values of V1, V2 and V3, respectively, from the instrument. The monitor then computes its value of V4, say V4', using formula F. The monitor then sends the command GET? V4 to the instrument to get the value of V4 from the instrument. The monitor then compares V4 with V4'. If they are different an error alarm is communicated to the test engineer, else the test ends.

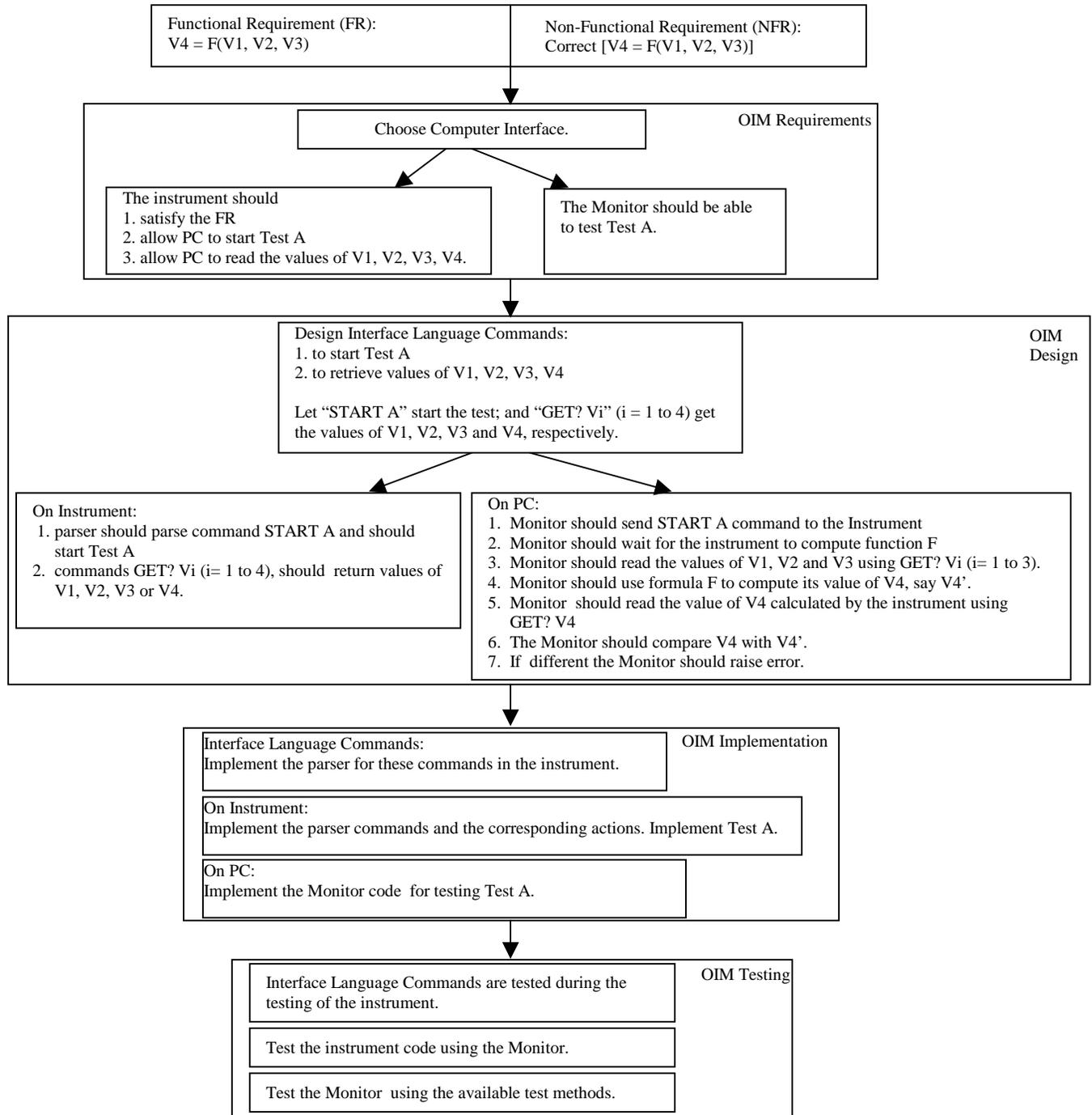


Figure 4: OIM Firmware Development Process Example

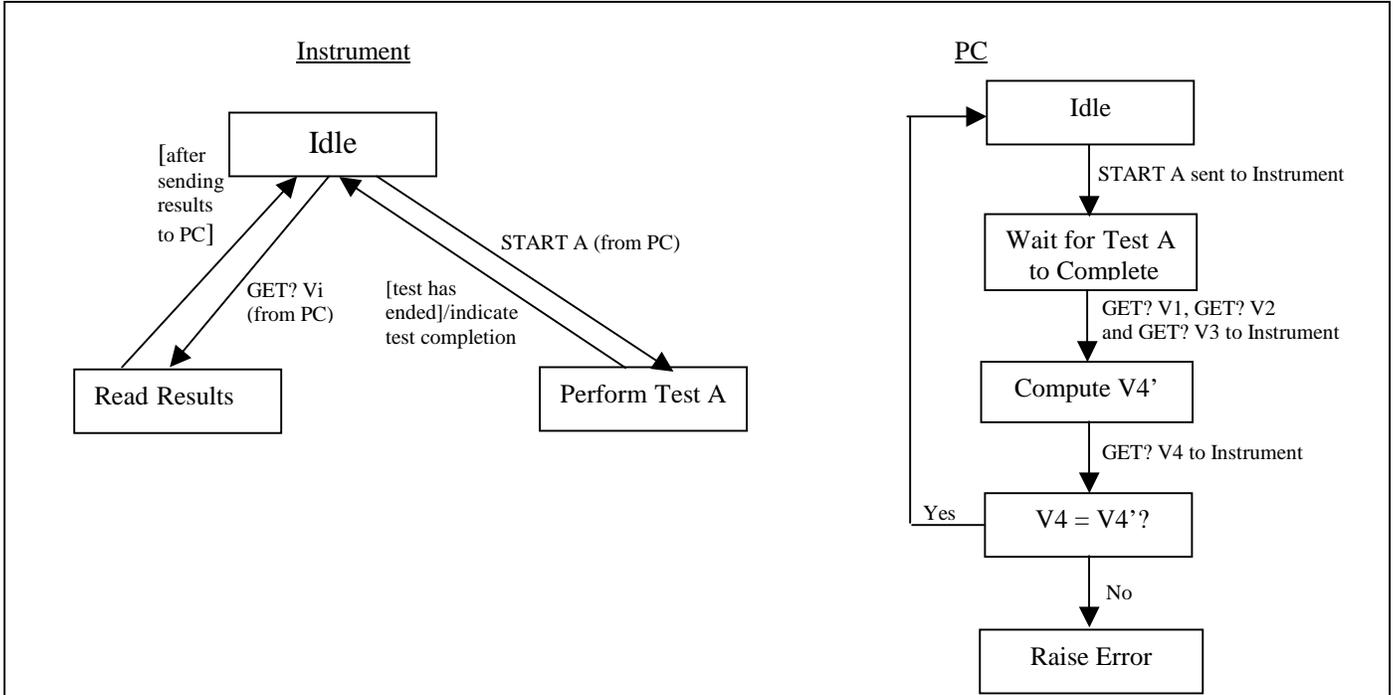


Figure 4a: STD's for PC and Instrument

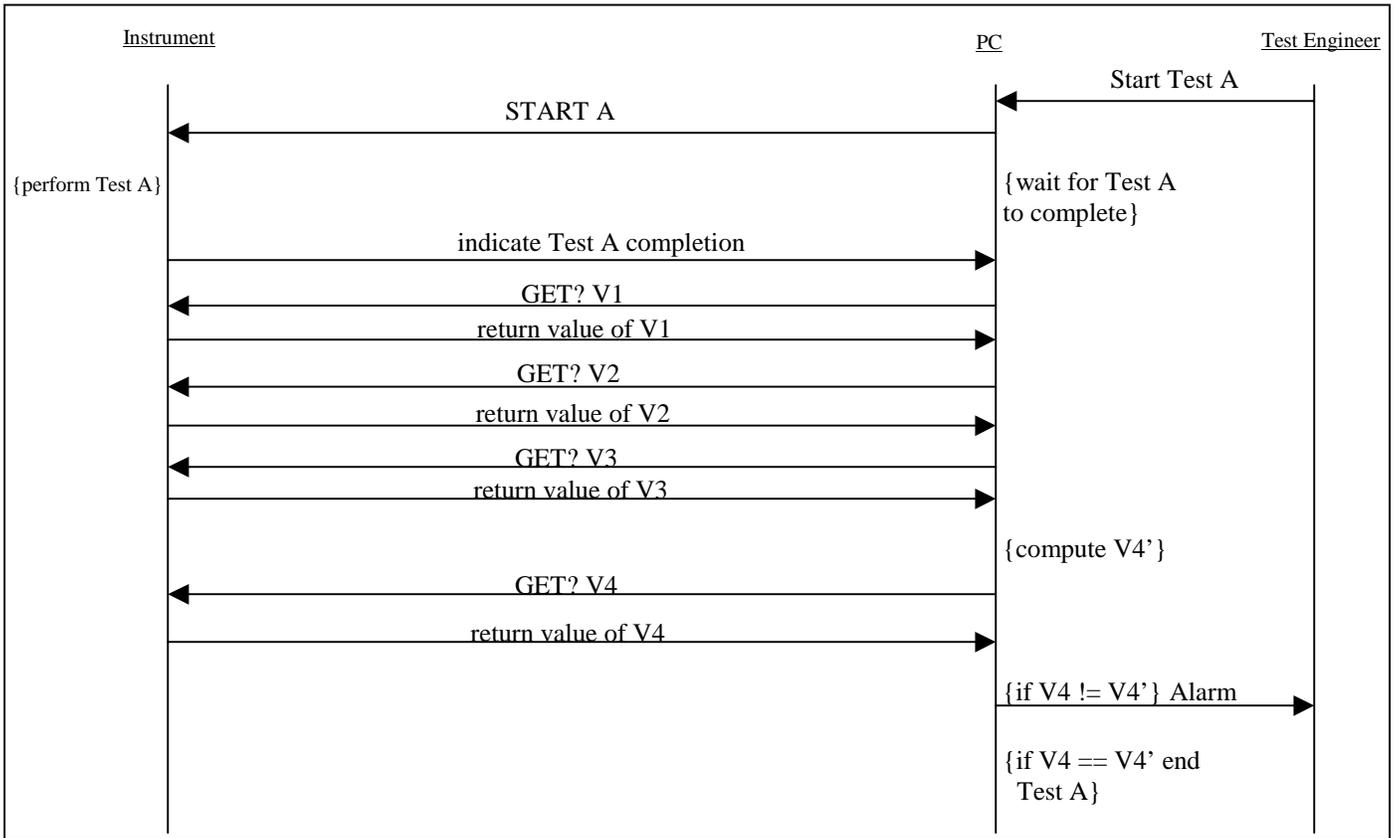


Figure 4b: Sequence Diagram for Test A

3.1.2 OIM Requirements

The requirements for OIM are the following:

1. A single “window” into all parts of the firmware is required.
2. This “see-all port” should be accessible by an external PC connected to the OIM system by a standard interface (USB, Firewire, IEEE488, Ethernet) – see Figure 1.
3. The design is oriented toward machine-machine testing.

In order that the above goals are accomplished, the following must be added to the requirements:

1. An interface driver
2. The Interface Language Commands
3. Monitor application on a PC.

3.1.2.1 Interface Driver

Interface is a hardware port to the system being developed. Thus this interface driver requirement should be part of system requirements also. This interface will be the “see-all port” mentioned above. However, while the interface is a hardware component, its driver is the software component. The driver lets the external world talk to the application running on the system and also lets the application on the system talk to the outside world. This driver should be part of the software requirements. It is quite possible that an interface was already a part of the system requirements; in that case the OIM requirement for the interface driver is not considered.

3.1.2.2 Interface Language Commands

The language is the tool with which the external PC communicates with the system. Since, every parameter in the system should be visible to the outside world, there should be a command that the external PC will send to the system to set or get each parameter. Thus this language requirement should be part of the requirements. Along with the language comes the requirement for its parser and this parser should also be part of the requirements.

3.1.2.3 Monitor

The monitor is the application on a PC that will be used for testing the embedded system. The monitor will send commands to the instrument and read data back from the instrument, and interpret the data received.

3.1.3 OIM Design

The design phase is same as that of the usual software development process with the exception that the requirements have been changed for OIM. Thus the design phase should ensure that the “window” into all parts of the system is created; should ensure that there is a command in the interface language to get or set each parameter in the system and should decide on the parser algorithm for the interface language commands.

Design Interface Language Commands. Different categories of interface language commands will be required and they are listed below:

1. Commands to Set the Context of the Instrument. Examples are:
GET READY FOR POWER MEASUREMENT
SET UP A CALL WITH PHONE
CHANGE TO DIFFERENT PHONE SYSTEM
2. Commands to Set Values of Parameters. Examples are:
SET OUTPUT LEVEL

SET FREQUENCY
CHANGE DELAY TIME

3. Commands to Change State of Parameters. Examples are:

OUTPUT POWER ON
SPECIAL MODE OFF

4. Commands to Start the Instrument to do tests. Examples are:

START POWER MEASUREMENT
EXECUTE FUNCTION F
START ANALYSIS MEASUREMENT

5. Commands to Retrieve Values/States of Parameters and Results. Examples are:

GET OUTPUT LEVEL
GET OUTPUT POWER STATE
GET MEASURED POWER VALUE

6. Commands to Set and Get Miscellaneous Parameters. Examples are:

SET TIME
SET DATE
GET TIME
GET DATE

The interface language commands for any embedded system can be divided into two classes – application independent and application dependent. Application independent commands are the commands applicable to almost any embedded system, while the application dependent commands are commands specific to an embedded system. The application independent commands can be designed from the state diagram of the generic embedded system given in Figure 5.

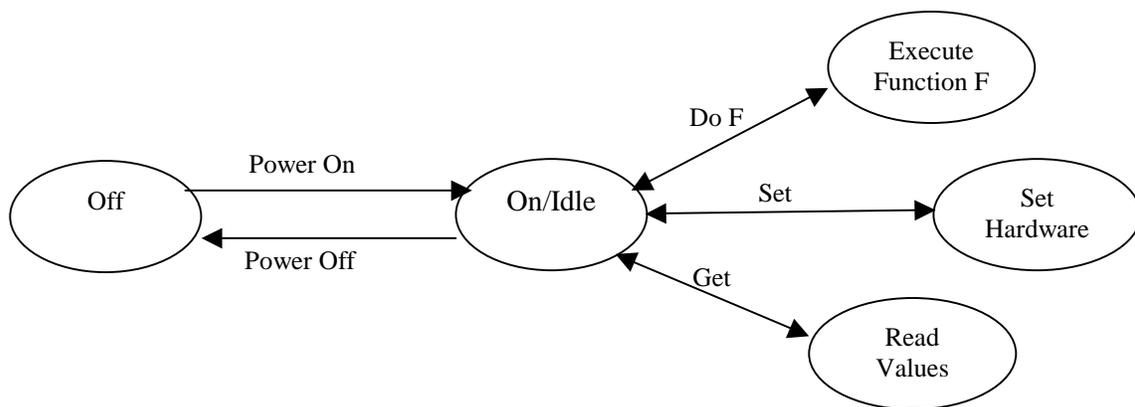


Figure 5: State Transition Diagram for a Generic Embedded System

As Figure 5 shows, the embedded system is initially in Off state. Upon pressing the power on switch the event Power On is generated which causes the transition of the system to the On or Idle state. Upon receiving the event Do F (which may be due to a key press or due to a command received over an interface port) the system goes from the Idle state to the Execute Function F state. In the Execute Function F state the system executes the function F and upon completion returns to the Idle state. When the system receives the Set event (this event will have at least two parameters – the parameter to set and the new value of the parameter) the system goes to the Set Hardware state where the parameter is set with its new value. After setting the system returns to the Idle state. Upon receiving the Get event (this will have at least one

parameter) the system goes to Read Values state wherein it reads the value of the parameter and returns to the Idle state.

Figure 6 shows the table of events in the generic embedded system and how commands are derived from the table.

Event	Description	Example Commands Generated
Power On	power is turned on; system goes to Idle state	
Do F	executes function F	START POWER EASUREMENT EXECUTE FUNCTION F
Set m,n	set value of parameter m = n	SET OUTPUT LEVEL SET FREQUENCY SET DELAY TIME OUTPUT POWER ON SET TIME
Get m	get value of parameter m	GET OUTPUT LEVEL GET MEASURED POWER VALUE GET TIME
Power Off	power is turned off; system goes to Off state	

Figure 6: Event Table used to Generate Interface Language Commands

Design Instrument. In designing the Instrument of the OIM system the advantage of reuse can be obtained for many cases. This is because the original system already had some means of doing the activity – for example, to start a test there would have been a key to press (this is the primary input). Then the code that the key press executes can be simply used by the interface language command that starts the test. This is reason for the small overhead for OIM implementation (see section 3.4). For reading values from the OIM system there has to be a memory of some sort to store the values to be read; this is because some values may be transient – produced and consumed within a very short interval of time and overwritten by the subsequent value. All these transient values will have to be stored in the memory so that after the test that caused these transients to occur has been completed, the PC can get these values out of the system for analysis. The parser for interpreting the Interface Language Commands should be designed and the driver to handle the inputs from and the outputs to the interface should also be designed. Figure 7 shows the design for the instrument of the OIM system.

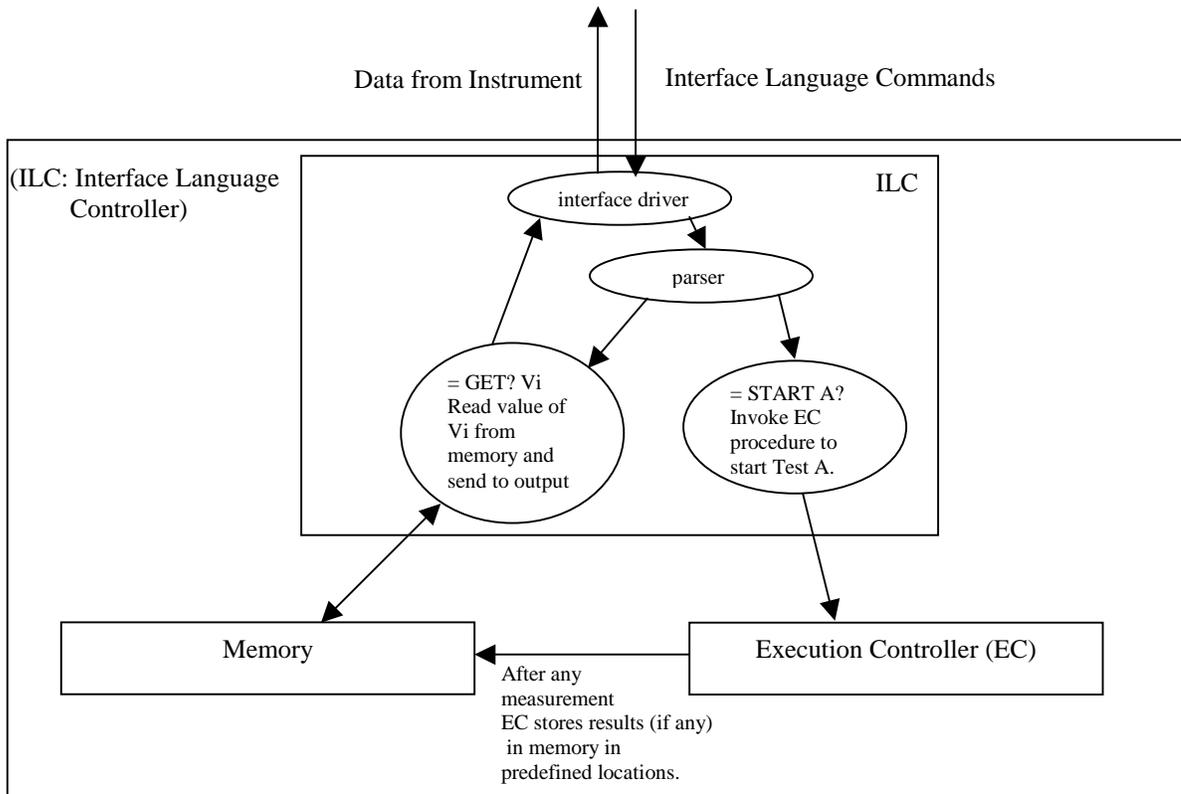


Figure 7: Design of the Instrument for the OIM System

Design Monitor. However, for the PC, the design phase involves the monitor design – the monitor should be designed for testing each and every function in the instrument. The generic pseudo-code for developing a test module in the monitor is given in Figure 8. All lines in Figure 8 with “(if necessary)” are optional and are not required for all tests.

At step 8 in Figure 8, the error is raised to inform the test engineer that the test did not complete successfully. If for testing function f , the steps 1 to 8 were done in a loop many times, then upon error the test will break out of the loop and will not continue.

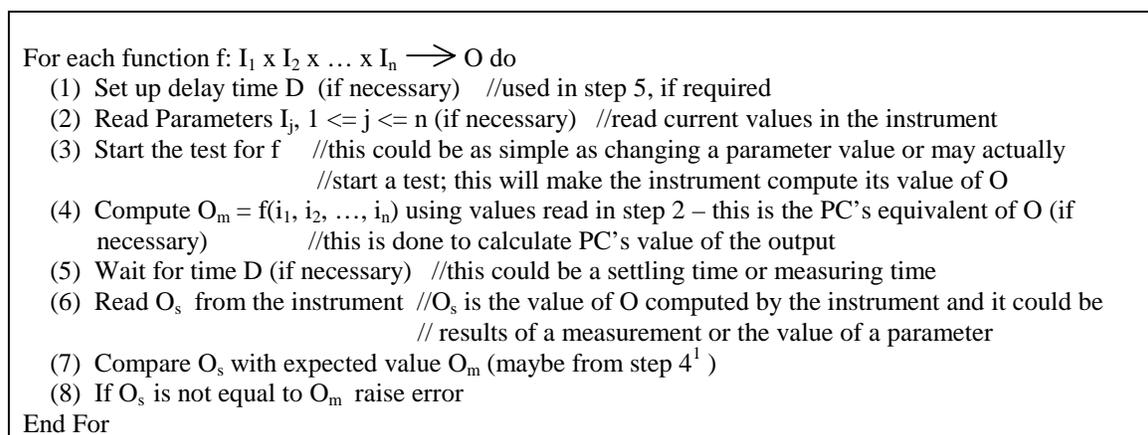


Figure 8: Generic pseudo-code for a test module in the monitor

¹ This is because O_m may not always be a computed value. It could be a fixed constant too in which case step 4 is omitted and the fixed constant is used in step 7.

3.1.4 OIM Implementation

The implementation phase implements the design, for both the embedded system and the monitor.

3.1.5 OIM Testing

The testing phase of the OIM process is totally different from the standard process. In OIM the monitor tests the instrument. The monitor sends the test commands to the instrument and reads responses from the instrument (responses are sent by the instrument to those test commands that require responses to be sent). It is very easy for the monitor to check the responses to see if they are expected or not. Moreover, the monitor could log the results of all the tests for human observation and regression tests.

3.2 OIM Rationale

Why should OIM require computer interface development be the first step of the firmware development process? This is because the design should consider the computer interface at every step. Since almost all internal functions and variables should be accessible from the interface, there must be some mechanism for the interface to see all the data – either global variables or message passing mechanisms could be used. And this can be done effectively only when the design starts with computer interface. If the computer interface part of the software is added as an afterthought, then the insertion of “access points” to the different parts of the software becomes a challenging activity and if not correctly implemented could cause the effectiveness of the interface in testing to decrease. The OIM obviates this risk. One of the popular software development approaches is the incremental development model. If OIM is used, every increment of the software will be tested through the interface. The very first increment may have only the boot-up software besides the interface software. The first increment will have to be exhaustively tested to make sure that the interface software works correctly (with proper stubs). Subsequently as more features are added to the software, more test commands will also be added and the newly added features can then be tested thoroughly. As can be seen, the OIM methodology complements the traditional software development process – in the latter, the computer interface (even if required) is rarely a part of the very first software version. The riskiest or the most critical part of the software is developed first [8]. However, there is no way to exhaustively test the software so developed. Whereas OIM is oriented toward automated testing of software right from the start.

3.3 Interface Types

So far the paper has referred to computer interface in a generic manner. There are numerous physical interfaces available like RS232C, IEEE488, USB, FireWire, Ethernet and the like. The interface could be serial or parallel, wired or wireless. Whatever interface the development group is familiar with can be used.

3.4 OIM Overhead

Since OIM insists on a computer interface one may be tempted to conclude that a high overhead penalty for the embedded system will be incurred. If the interface was not a requirement for the instrument then additional software is required for the interface driver, command parser and command execution. In the author’s experience, if software reuse is followed, then the overhead does not exceed a few tens of kilobytes (in the project that the author was involved, the interface software was less than 27 kilobytes). Since modern embedded systems often have a 32-bit CPU and megabytes of memory [9], the interface overhead should not be a constraint. If, on the other hand, the interface was part of the system requirement then the additional overhead will be very low indeed.

4. IMPLEMENTATION

One of the authors was involved in a project that used OIM for the most part. The embedded system developed was a high-end telecom test and measuring instrument. The computer interface used was IEEE488 (also known as GPIB) that is a parallel interface. One of the advantages of GPIB is that it has a dedicated service request line and this was used extensively for testing purposes. A PC with a GPIB-port was connected to the instrument using a GPIB cable. The monitor was developed and run on the PC.

4.1 Implementation of the software

The software that was implemented included the complete stack on the instrument side and the application layer on the PC side as shown in Figure 9.

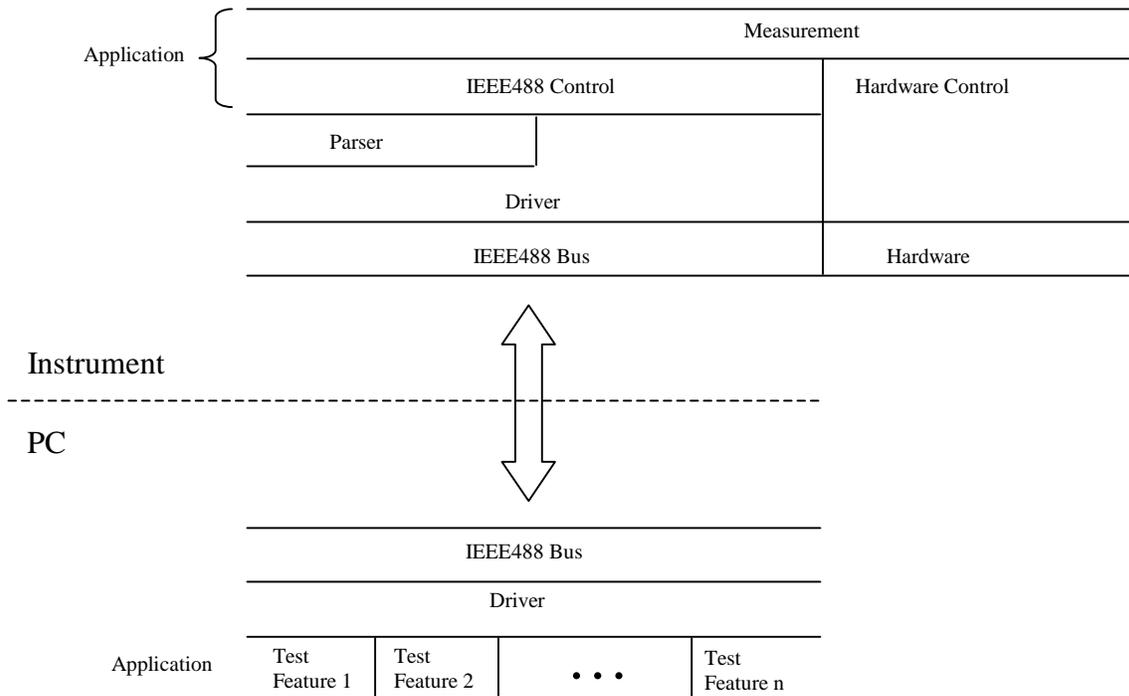


Figure 9: Software Architecture Used for Implementing OIM

On the PC side, the Driver and IEEE488 Bus are third party software and hardware, respectively. The Application layer on the PC is the monitor that runs different tests on the instrument. The test code on the PC sends out a series of commands to the instrument and to some of these commands it waits for a response from the instrument. The IEEE488 standard differentiates the commands that wait and don't wait for instrument responses by a command that terminates with a '?'. If a command terminates with '?' it means that the monitor expects a response from the instrument, else the monitor doesn't. The PC is connected to the instrument by an IEEE488 cable.

On the instrument side, the IEEE488 Bus layer is the hardware layer and is handled by an ASIC. The hardware-software interface is handled by the Driver layer. This layer sends outputs to the hardware layer and receives inputs (in the form of interrupts) from the hardware layer. The Driver layer then sends the commands from the PC (i.e., the inputs to the instrument) received from the hardware layer to the Parser layer. The Parser layer decodes the commands and sends the decoded message to the Application layer. The Application layer consists of the IEEE488 Control layer and the Measurement layer. The IEEE488 Control layer takes the correct action based on the commands. If the command requires only an action to be taken by the instrument like changing a parameter value or doing a measurement, the IEEE488 Control sends the appropriate messages to the Measurement layer to take these actions but does not send an

acknowledgment to the PC; but if the command needs a response to be sent back to the PC, the IEEE488 Control layer generates the response by reading the necessary values from the Measurement layer and sends the response back to the Driver directly (the Parser layer is not involved in the return path), which then sends the response onto the PC via the hardware layer.

For example, if the command “TAKEMEAS” requires the instrument to take power measurement, say, then when the PC sends this command (this command may be part of a “Test Feature i” module of the monitor), the hardware layer of the instrument receives the command “TAKEMEAS” byte by byte (IEEE488 is a parallel bus) and sends the full command to the Driver layer. The Driver layer resets the hardware layer for receiving/sending subsequent commands from/to the PC and sends the received command to the Parser layer for processing. The Parser layer decodes this string to mean “do_power_measurement” and calls this function in the IEEE488 Control layer, which then sends the message to the Measurement layer to do the measurement. If the PC wants to read the result of this measurement, the PC then sends the following command “RESULT?” (for example) and the Parser layer, this time, may decode this string to mean “return_measured_value” and calls this function in the IEEE488 Control layer. This function in the IEEE488 Control layer will read the result from the result-area of the Measurement layer and send the response (for example, 5 DBM) directly to the Driver layer. The Driver layer sends this response to the hardware layer. In IEEE488, the PC takes an explicit action to read the response and as soon as the instrument’s hardware layer knows that the PC wants the response, the former sends the data in the output buffer to the PC. The PC can now store such results in a big array to plot them or do any other manipulation with the data. The fact is the data in a running system (the instrument) has been received in the PC where further (and perhaps faster) processing can be done.

4.2 Pseudo-codes for the Monitor

Many examples have been presented earlier on how the testing is done in an OIM-based system. This section presents even further examples with pseudo-codes to clarify this point. Monitor is the application that is resident on the PC and tests the code running the application inside the embedded system (i.e., the instrument). The monitor is written in C language (it can be in any language) and has modules to test various parts of the system.

An example code is given in pseudo-code below that tests the following scenario: Upon setting the output level for the instrument, say to 10dBm (= 10mW), the instrument sets the registers of its internal hardware to set this level. Three hardware registers have to be set for any level change. The registers are entered integer values that are calculated from the set level by somewhat complicated formulas. Let these formulas be F1 for setting the first register, F2 for setting the second register, and F3 for setting the third register, and let the three register values be R1, R2 and R3, and let the set output level be L. Then

$$\begin{array}{ll} R1 = F1(L) & \dots (1) \\ R2 = F2(L) & \dots (2) \\ \text{and, } R3 = F3(L). & \dots (3) \end{array}$$

Since, the R1, R2 and R3 values are internal, they are not accessible to the user of the instrument. However, in the OIM method, these values are accessible through the test port and let GET? R1, GET? R2 and GET? R3 be the commands for accessing these register values, and let “SET L,f” be the command to set the value of L to f (since the test port is used for testing and this testing is only performed by the company that manufactures the instrument, there is still no need for the customer to be aware of the commands to retrieve R1, R2 and R3 values – he may need the SET L,f command though, in case he chooses to use this port for setting the value²).

²This is because the requirement for an external PC interface port may have been part of the original requirements. In that case, it may be simpler to choose the same interface to be the OIM test port as well. However, in this case while SET L,f may be the command that the user will be informed about (so that he may set the output level remotely), he will not be told about the GET?R1, GET?R2 and GET?R3

Then the pseudo-code for the monitor will be as given in Figure 10. This pseudo-code is not very different from the generic code given in Figure 8. Here the separate tests for R1, R2 and R3 have been collapsed into one single test.

```

DO FOR ALL OUTPUT LEVELS f from f1 to f2:
  SET L,f          //set the value of the output level in the instrument to f
  CALCULATE THE VALUES OF R1, R2, R3 USING (1), (2) and (3).
  r1 = GET? R1    //get the value of R1 calculated by the instrument and set it to r1
  r2 = GET? R2    //get the value of R2 calculated by the instrument and set it to r2
  r3 = GET? R3    //get the value of R3 calculated by the instrument and set it to r3
  IF (r1 NOT EQUAL R1) DECLARE R1 SETTING ERROR AND EXIT
  IF (r2 NOT EQUAL R2) DECLARE R2 SETTING ERROR AND EXIT
  IF (r3 NOT EQUAL R3) DECLARE R3 SETTING ERROR AND EXIT
END DO

```

Figure 10: Pseudo-code for the Monitor (first example)

The above is a simple example. However, let us consider the case in which the hardware registers had to be set within 100ms of receiving the command to set the output level and that there was a timer that kept track of how fast the output levels were set after receiving the command to do so (this command may have been received from the remote PC or may have been set by the user from the instrument's front panel). In the OIM methodology, let `TIMER?` be the command to retrieve the value of the timer after each setting (it is assumed that the timer is reset every time the output level is changed). Then the pseudo-code in this case will be as given in Figure 11.

The typical time for sending the command `TIMER?` and reading its response is about 10ms (using IEEE488). Hence, the PC can know the timer values much faster than, say, if the timer values were printed on the instrument's screen (if it were at all possible) and were being manually interpreted. Also, in the second case above, there is no way to test the code passively and know whether the output level will always be set within 100ms. This is because there could be unexpected hardware-software interactions in the running system that is completely ignored in the passive method of testing. This is where the power of OIM lies.

```

DO FOR ALL OUTPUT LEVELS f from f1 to f2:
  SET L,f          //set the value of the output level in the instrument to f
  CALCULATE THE VALUES OF R1, R2, R3 USING (1), (2) and (3).
  WAIT FOR 100ms  //so that the maximum time for setting registers is over
  time = TIMER?  //get the timer value and put it in time (and the timer is reset)
  IF (time GREATER THAN 100ms) DECLARE TIMER ERROR AND EXIT
  r1 = GET? R1    //get the value of R1 calculated by the instrument and set it to r1
  r2 = GET? R2    //get the value of R2 calculated by the instrument and set it to r2
  r3 = GET? R3    //get the value of R3 calculated by the instrument and set it to r3
  IF (r1 NOT EQUAL R1) DECLARE R1 SETTING ERROR AND EXIT
  IF (r2 NOT EQUAL R2) DECLARE R2 SETTING ERROR AND EXIT
  IF (r3 NOT EQUAL R3) DECLARE R3 SETTING ERROR AND EXIT
END DO

```

Figure 11: Psuedo-code for the Monitor (second example)

commands. These latter commands were developed and implemented only as part of OIM. Likewise, the `TIMER?` command for the second example may be an exclusive OIM command.

Thus, as can be seen the interaction between the monitor application and the original requirements is through the commands sent between the PC and the instrument. There is no other way for these two independent applications to interact. All the interaction must be through the test interface chosen.

5. RESULTS OF USING OIM

The embedded system that was developed was a test and measuring instrument that tested cell phones before release to the market. The OIM was used for most part of the development of the firmware for this embedded instrument. The embedded system took about ten engineers more than a year to develop. Such test and measuring instruments are used by cell phone manufacturers and service providers. Before OIM was actively used, there had been many recurring complaints on the stability and performance of the system. Stability related to the robustness of the system – the system should not crash in the presence of reasonable user interactions; performance related to the working of the instrument over time – previously the system used to crash after working some x hours but such transient bugs were ironed out using OIM. Since OIM has been used, however, the customers have informed that the subsequent versions of the firmware have been better both in stability and performance. This has led the company to feel more confident that the firmware releases are error-free.

Currently this project is in its final phase. Customers have been satisfied with the features provided so far and the performance of the instrument. The IEEE488 interface was part of the system requirements. For OIM the same interface was used. In all about 1000 interface language commands were developed, out of which about 20% were for OIM purposes and the remainder were for meeting customer requirements.

The monitor tested all aspects of the firmware. This resulted in reduced testing and error detection times. Early detection of the error meant that the development group could fix the bugs before the release reached the customer. OIM helped detect numerous bugs that could not be detected in any other way (some tests were run 200,000 times to detect transient bugs – one test run lasted one second and the tests were run 3-4 days). One of the examples of OIM's advantage was clearly evident in a test (which shall be referred to as Test A – this is the same example that was used earlier) that spewed out four values, say V1, V2, V3 and V4, where V4 depended on V1, V2 and V3 by a formula F, i.e.,

$$V4 = F(V1, V2, V3).$$

As soon as Test A completed the instrument raised the service request line, whereupon the PC read the values of V1, V2, V3 and V4. The PC used the formula F and computed V4' (the PC equivalent of V4). When the Test A was run thousands of times it was found that there was a significant difference between V4' and V4 in some cases. To further analyze the problem, the intermediate values that the instrument used for formula F (there were 3 of them, say I1, I2 and I3) were also retrieved by PC using special test commands developed only for this purpose. When I1, I2 and I3 were used, the values of V4' and V4 agreed exactly. The problem was then found to be due to truncation of floating point values of V1 and V2 when read by PC (the instrument used more precise values). The PC ran test scripts written in C to do the tests and gave the results of the test in a spreadsheet format. This enabled easy analysis of the results including chart creation.

6. CONCLUSION

Through numerous examples, this paper has presented the Out-In methodology (OIM). This paper has also presented a real application of the methodology, which detected numerous functional and performance-related errors at run-time, hence enhancing the stability and performance of the application.

It is the contention of the authors, drawn from this application, that very little or no organizational change is required to implement OIM. In fact, once followed, the popularity of the OIM methodology will most likely increase in any organization, although further studies would be needed to confirm this generalization.

OIM has its drawbacks too. Since an external PC is being used to send commands to and receive responses from the embedded system, the embedded system will have to service the interrupts from the external PC. This takes up processor time in the embedded system that could affect the time taken to complete the other processes running in the system. However, in practice, this is not that much of a constraint, since once a measurement is started the PC waits for the measurement to complete before reading the results. Thus while the embedded system is doing the measurement, it is not disturbed. But this means when selecting the processor for the embedded system the time spent in processing the PC interrupts should also be considered. Another drawback is that since the OIM methodology requires testing of the system's function, the test code on the PC may duplicate most of the code that the embedded system uses. This may be a constraint in some cases. Yet another drawback is the possible occurrence of race condition – a test may have been started by the PC, but due to some firmware error in the embedded system the test may never finish. But the PC may timeout waiting for the test to complete and may start reading the results of the (yet to be completed) test. Manual intervention may be necessary to stop the test and restart it after fixing the firmware errors. And finally, the OIM requires additional memory for storing intermediate values, as explained in section 3. However, in our experience the additional memory requirement was not a constraint.

It is the contention of the authors that the future of the OIM methodology is pretty promising. In this era of internet and anytime, anywhere access to the web, if an embedded system is equipped with web server capability, it can be tested from practically anywhere in the world if OIM is used. Thus customer service would acquire a special meaning – service by remote. The entire diagnosis can be done remotely and only if it is a hardware problem does a service engineer need visit the customer; if it is a software problem, even a new firmware version with the necessary fixes may be downloaded into the system remotely. However, this should not divert attention from systematic software development – if the scenario analysis is done upfront thoroughly then many errors in logic can be prevented. The OIM methodology can then be used to test the runtime firmware behavior most of the time and to test less of the mundane logic and coding errors.

REFERENCES

- [1] P.Lettieri and M.B.Srivastava, "Advances in Wireless Terminals", *IEEE Personal Communications*, February 1999, pp. 6 – 19.
- [2] P.A.Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, *IEEE Computer Society Press*, 1993.
- [3] R.E.Eberts, *User Interface Design*, Prentice Hall, 1994.
- [4] P.J.Byrne, "Reducing Time to Insight in Digital System Integration", *Hewlett-Packard Journal*, June 1996, Article 1.
- [5] E. Kilk, "PPA Printer Firmware Design", *Hewlett-Packard Journal*, June 1997, Article 3.
- [6] R.S.Pressman, *Software Engineering*, McGraw Hill, 1997.
- [7] M.Fewster and D.Graham, *Software Test Automation: Effective Use of Test Execution Tools*, Addison Wesley, 1999.
- [8] B.P.Douglass, *Doing Hard Time*, Addison Wesley, 1999.
- [9] P.Varhol, "Internet Appliances Are The Future Of Small Real-Time Systems", *Electronic Design*, September 20, 1999, pp. 59 – 68.
- [10] R.Lewis, D.W.Beck and J.Hartmann, "Assay – A Tool To Support Regression Testing", *Proceedings of 2nd European Software Engineering Conference*, 1989, pp. 487 – 496.
- [11] N.Subramanian, "Instrument Firmware Testing Using LabWindowsTM/CVI and LabVIEWTM – a Comparative Study", *Proceedings of NIWEEKTM*, August 1999.
- [12] N.Subramanian, "A Novel Approach To System Design: Out-In Methodology", presented at *Wireless Symposium/Portable by Design Conference* in Feb., 2000.