# An Investigation into Improving Test Effectiveness for Embedded Software

Department of Computer Science
National University of Ireland Maynooth Campus

Name: Greg Thomas

January 2009

## Declaration

## Abstract

This thesis reports on the investigation of the effectiveness of software testing on embedded systems. The aim was to improve confidence in the current methods employed or to find new methods which could improve the hit rate of defects found before software is sent to a customer. We investigate previous work into software testing effectives and various black box testing methods. There are various Black Box testing methodologies that can be employed to detect errors in systems with varying degrees of success. In this thesis we investigate the transformation of the white box testing technique of Definition Use (DU) Path testing using a RESOLVE like specification, to be applied as black box test method. We do not use RESOLVE it self, instead we defined our own method of automatic test generation based on the principles of RESOLVE. Then we compare this method to more commonly used requirements driven test selection, and pure boundary value analysis (BVA) testing techniques. The results reported in this thesis indicate that BVA and DU test selection methods create tests that are covered by unit and integration tests. The current requirements driven test cases create tests with a combination of features working in tandem. It was found that combination of features was more likely to find defects because developers tests had a lesser focus on this area. The tests generated by the BVA and DU test selection methods did not find any defects that their respective methods were intended to find. This is due to the development team already having tests that covered these areas and defects had been fixed before system tests could be run. Based on the fact that the current test selection methods find defects and the methods we investigated do not, this adds confidence that the system test approach to testing is effective. The investigation of defects found showed that timing related errors are common and that a test selection method designed to find timing related defects would be worth investigating. The investigation also revealed a useful method in automatic generation of test cases. The RESOLVE like specification was used to apply a DU testing as a black box test method. This method showed to be more time efficient at generating test cases than the existing requirements driven approach. Although the test cases did not reveal significant defects, due to the overlap with integration testing, it could be a useful method for developers to generate test cases.

# 1 INTRODUCTION

## *1.1 Problem Statement*

The author hopes to improve upon or add some confidence in the test case selection process currently used at the author's place of employment. The quality of the current method (described in section 4) is rather adhoc in the test generation but believed to be effective, based on the fact that it finds defects. However it is good software practise to try something different in case there is a better test selection method that either finds more defects or is more efficient in generating test cases.

This work investigates the effectiveness of various black box testing methodologies against embedded software developed by the author's employer. The existing method of test selection is essentially a requirements driven selection of test input which has some elements of boundary value analysis (BVA). The BVA elements are purely driven by the need to validate maximum performance of the system which implies input of maximum values to the API. Otherwise test case input is driven by the need to validate requirements (this is discussed further in section 5.3). Test case selection is adhoc with some additional effort made to vary the test input to cover what is perceived to be common customer configurations.

The cons of the current method of system test selection method are listed as follows:

- test code is designed and implemented manually, naturally this is a time consuming process
- The tests are requirements driven and do not naturally fit into an automated test framework. The test Framework is only designed to cover positive test cases and does not handle negative or abnormal situations. For example it is not possible to automate a test were a cable is physically removed from the system. This makes test execution and automation of tests a costly/time consuming process.
- Tests depend heavily on the use of external traffic generators, this new equipment can take time for users to get familiar with and there is additional

work in automating this equipment. (Automated test equipment saves time later on when used for repeated regression testing)

- Test case selection by the system testers is similar to test case selection by the integration testers, leading to an overlap of tests between integration testing and system testing

The pros

- The method finds additional defects that the integration tests don't find, even though it is similar to the integration testing and 100% of integration tests have been executed and passed.

Fundamentally there is nothing wrong the current method of test case selection. It takes time but appears to be effective; however effectiveness can only be measured when compared to other processes that aim to achieve the same thing. Based upon the learnings gained from the modules taught in the MSc in Software Engineering course at Maynooth, it is appropriate to review the effectiveness of the current test case selection method by comparing it to other well documented methods.

## *1.2 Background of the Investigation*

As part of the requirements of completing the MSc in Software Engineering at NUI Maynooth, a student is required to submit a thesis based on the software engineering practices which occur at the students work placement. The author undertook the course part time whilst working from Intel Shannon as a system test engineer and therefore this thesis is based on work carried out at Intel Shannon following the study of all modules in the MSC in Software Engineering course.

The Authors role at Intel Shannon as already mentioned was a system test engineer. This role involves the validation of Software written by Intel, to ensure that it meets the software product requirements before being made available to customers. The software is aimed at enabling the use specialised network processors. A normal software release will contain a series of drivers and hardware access layer software components in which the customer can use in their own applications. The requirements of this software is derived from customer requests and is generalised to cover many possible customers. The software released is free to download from Intel's website. The quality of the software released is dependent on its use. Sample code can be released but customers are not expected to use it in there own product so it does not require the same level of testing of code that customers will put into there products.

This thesis reports on the investigation of the current system test techniques used by the system test and validation team at Intel Shannon and compares it to other black box testing techniques to see if the current method being employed is effective for finding defects.

## 1.3 Software Testing

This section describes the test stages from unit testing, integration to system testing performed and how the company differentiates the stages.

### 1.3.1 Unit/Component Testing

Unit testing is also referred to by some as component testing. Beizer[8] defines component testing is an aggregation of one or more units that can be compiled together as a component, based on this definition, the lowest level of tests performed by the company is component testing. Component test code performs boundary-checks on all API interface parameters and checks pre and post conditions, as specified by the API documentation.  On execution, it will attempt to exercise as many paths through the production code as possible with an objective goal of 100% functional coverage 80% decision/branch coverage. A tool called Bullseye is used to measure coverage, and then tests are added until the required code coverage is achieved. Sometimes it is not possible to achieve 100% coverage, for example a condition may occur that code is only executed in the case of a hardware malfunction. We do not want to damage the hardware just to cover an unlikely scenario. The Bullseye tool is not available to use at the system test level.

Stubbed functions are written for any module that the component is dependent on. The stub simulates the dependent module and returns messages as per the specification of that stubbed API.

The development team are responsible for component testing. Test case input is selected manually to cover all boundaries and return value checks and achieve the coverage metrics listed above.

### *1.3.2 Integration Testing*

Beizer [8] defines integration testing as a process of testing an aggregation of components to create larger components. At Intel Shannon the interpretation of integration testing is similar – components are tested to work with other components. In component level testing, components are tested as one component with stubbed versions of dependent modules. Integration tests rely on fully functional dependent modules and the focus is not on boundary value checking or return message checking but rather that the components work together properly.

The development team is responsible for the implementation and execution of integration tests. Their mission is to release software to the system test team that is below a software quality threshold – expressed as defects per thousand lines of code. The threshold is set by the software quality engineer and may vary from project to project. The development team also need to ensure that all product requirements are met, so their test case selection method is requirements driven.

### 1.3.3 System Testing

System testing is aimed at ensuring all components work together as a system. Testers should aim to test all possible combination of interfaces between components, with the aim of exposing defects that could only be exposed by testing all components together. Depending on the size of the system this could overlap with integration testing if the number of components in a system is small.

The system test team approach the system as a black box. Tests are developed based on a product requirements document using the method described in section 5.3, the high level design document and the API of the software to be provided. Quite often new requirements are implied by the API and design documents so it is important to check that tests cover features listed in all documents. Test case selection is requirements driven; the result of this is that system level tests and integration level test can be very similar, but done by an independent team. Beizer [9] does not believe in independent testing, it is a statement of mistrust on the developers, however he goes on to say that network testing is a specialized field and it is too much to ask the developer to be both productive and have an expert knowledge of networks. This analysis seems to be true for Intel; the independent test group approach has always found defects, despite the similarities in approach to test selection

### 1.3.4 Common Black Box Testing Techniques

This section describes some of the most common Black Box testing techniques. These techniques had not previously been used to generate test cases with the Intel Shannon offices.

#### 1.3.4.1 Equivalence Partition Testing

Equivalence Partition, Roper [17], testing is one of the basic forms of Black Box testing. It applies test data to cover each input and output of an API at least once. Each input and output parameters are divided into partitions, which according to the specification, are treated identically. It should not matter if the value is a minimum or a maximum value within the partition; the method assumes that the implementation processes them in a similar fashion. The coverage criterion is to ensure that each partition, be it an input or an output from a function, is covered in at least one test in a suite of tests. Invalid data may be chosen to ensure that the system under test handles the data correctly.

The strength of this method is that it helps to minimise the number of test cases generated and focuses on testing the specification. The weakness is that a while a specification may treat an input or output domain as equal, internally this may not be that case and it does not exercise the boundaries of inputs were mistakes can easily be made in implementation such as >= programmed as > or == as =.

### 1.3.4.2 Boundary Value Analysis

Boundary value analysis, Roper [17], is similar to equivalence partition testing except that it focuses on an area that is renowned source of faults – the boundaries of inputs and outputs. The inputs and outputs values of an API are partitioned and their range is also used as a means of selecting inputs. The maximum and minimum values for each partition are chosen and should be used as test vectors at least once in each test case as a coverage criterion. This will exercise the boundaries with the API.

The weakness is that it is very similar to equivalence partitioning, whilst it is better to find boundary related defects. The specification may not identify internal boundaries within a partition where data is handled differently.

### 1.3.4.3 Truth Tables

This is another specification based black box techniques also known as Cause & Effect Graphing, see Roper [17]. The method looks at the creating test input that stimulates the system – the "cause" of stimulus and analyses the output – the "effect" of input. The causes and effects are tabulated in a truth table expressed as statements that can be only true or false. The test is then able to create combinations of cause and effect statements. Some combinations will not be possible due to constraints in the system. Each combination of cause and effect statements is converted into test code that invokes the cause-effect combination under test. The expected output is outlined by the true of false of the effect statements.

The strength of this method is that it tries combinations of tests that might otherwise not have been tried. The method also creates the expected output as part of the process. The weakness of the method is that a large number of causes and effects can result in a large number of combinations that can become complex and time consuming to implement.

# 2 EP80579 Embedded Software Project

The EP80579 Embedded processor is an Intel Architected (IA)processing core with built in micro engines and a Programmable IO Unit. The micro engines and Programmable IO Unit can be programmed with firmware that allows the CPU to offload certain functions to the micro engines and Programmable IO Unit. Intel provides firmware and IA drivers that allow the offloading features to be enabled and also standard peripheral devices to be used such as SATA, Gigabit Ethernet, and USB just to name a few. The features that it can offload include: encryption and decryption of data streams, fragmenting data from the IA core onto TDM timeslots, channelising data received on timeslots to be sent to the IA core, adding/removing protocol headers, setting network byte ordering and processing bit endianess.

The software is provided to customers in the form of a binary executable for micro engines and Programmable IO Unit and source code for Linux kernel and user space drivers executed on the IA core. Customers compile and load the drivers into the IA core and this allows access to the micro engine and Programmable IO Unit offload features via IA kernel and users space drivers.

## 2.1 EP80579 IP Telephony Software

The EP80579 IP Telephony Software provides Programmable IO Unit firmware and IA drivers to allow the processing of PPP data and voice traffic. Given that the EP80579 also provides a Gigabit Ethernet interface and mezzanine card drivers to convert TDM T1E1 and POTS (Analog) onto TDM this allows the potential for the device to act as a media gateway between IP, TDM-T1E1 lines (ISDN) and Analog lines. The IP Telephony infrastructure is illustrated in Figure 1 below and can also be found in the Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology Programmer's Guide[21]

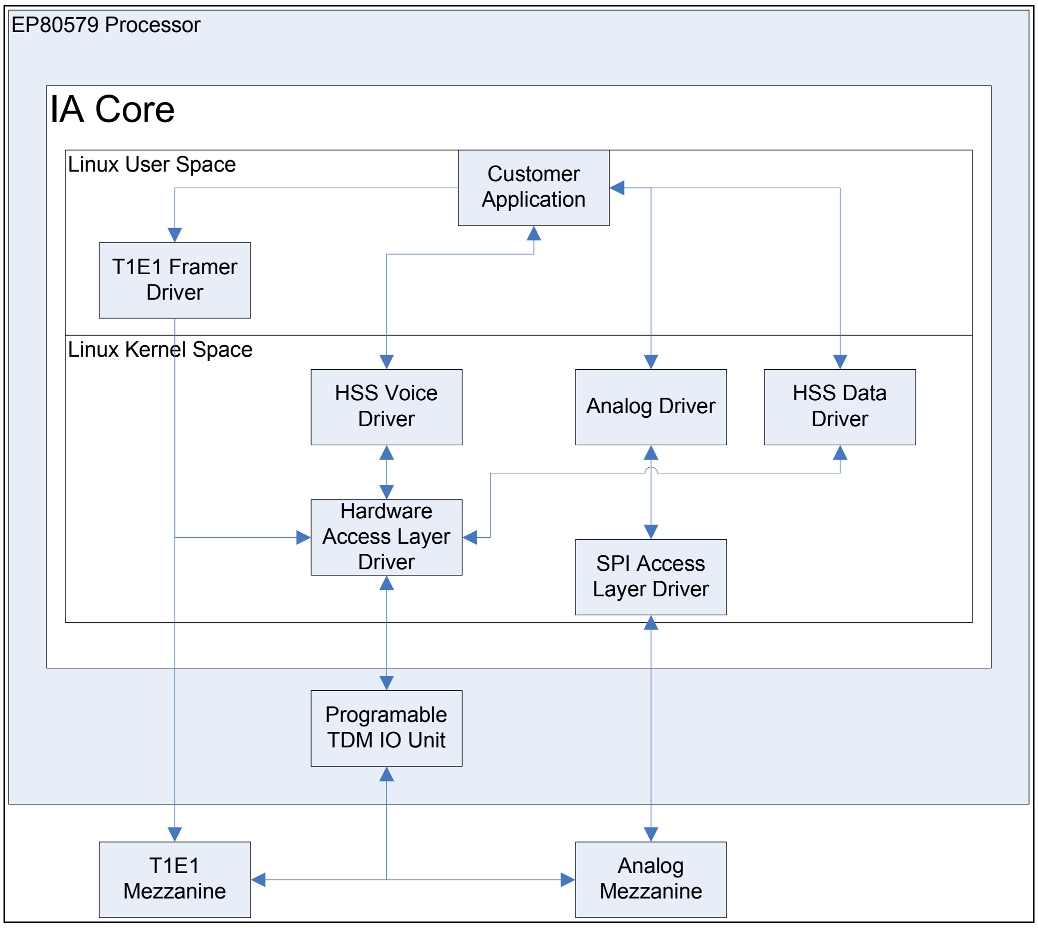


**Figure 1 EP80579 IP Telephony Infrastructure**

In this thesis our investigation focuses on system testing the functionality provided by HSS Voice Driver when used with the T1E1 Framer. As can be seen in Figure 1, the HSS Voice Driver is dependent on the Hardware Access Layer driver and either the T1E1 Framer driver or the combined Analog and SPI Access layer driver to provide the input and output of a voice stream.

The T1E1 Mezzanine is an add-on PMC-PCI card that provides for up to 4 T1 or E1 TDM Interfaces. Parikh, Keyur, Junius [3] provides an insight into the use of TDM networks and how Analog phone networks are gradually being switch to VoIP networks, the EP80579 processor is designed for this purpose. Each E1 Interface can carry 31[1] channels, therefore a single mezzanine using 4 E1 links are carry 124 channels – or 124 simultaneous voice conversations. Each T1 interface can carry 24 channels; therefore a single mezzanine using 4 T1 links can carry 96 channels – or 96

[1] E1 has 32 timeslots, but the 1st timeslot is used for signalling, leaving 31 timelots for voice traffic

simultaneous voice conversations. E1 is a European standard whilst T1 is a North American standard.

The Analog Mezzanine is an add-on PMC-PCI card that provides 4 FXS and 1 FXO port. An FXS port is what you would normally plug your phone into the wall at home. And FXO port is like the port at the back of your phone.

The system allows for up to 3 Mezzanine cards to be plugged into the system, this could be a combination of Analog, T1 or E1 or could be all the same. In this investigation we are testing the HSS Voice driver which has a requirement of supporting 128 channels, which can be done on 2 E1's therefore we choose testing with the Framer Driver to just configuring E1 Mezzanines on the system.

### 2.1.1 HSS Voice Driver

The HSS Voice Driver is a kernel level driver. It conforms to a Linux character device driver model. It initializes and manages voice channel communication with the Hardware Access Layer Driver. The customer application accesses the different voice channels managed by the HSS Voice Driver via the standard Character Driver operations (open, ioctl, read, write, close). A single file descriptor is used (/dev/hss-voice), and the different channels are multiplexed within the driver. Clients can access multiple channels per file descriptor if they so wish. For example, a single threaded client may wish to access multiple channels on one file descriptor, whereas a multiple threaded application may wish to access multiple channels by having multiple file descriptors which are each used to access a single channel. The HSS Voice Driver API is described in Appendix 0

The API has been summarised here:

- Open – opens the /dev/hss_voice device, the device may be opened up to 128 times
- ICP_HSSVOICEDRV_PORT_UP - ioctl command to bring up the port.
- ICP_HSSVOICEDRV_PORT_DOWN - ioctl command to bring down the port

- ICP_HSSVOICEDRV_CHAN_ADD - ioctl command to add and configure a voice channel.
- ICP_HSSVOICEDRV_CHAN_REMOVE - ioctl command to remove (delete) the channel, specified by the channelId in the parameter.
- ICP_HSSVOICEDRV_CHAN_UP - ioctl command to enable data flow on the channelId, specified by the channelId on the parameter.
- ICP_HSSVOICEDRV_CHAN_DOWN -ioctl command to disable data flow for the channel id specified in the parameter.
- ICP_HSSVOICEDRV_CHAN_BYPASS_ENABLE - ioctl command to create a unidirectional channel bypass between the channels specified in the data structure of type icp_hssvoicedrv_channelbypass_s passed as parameter.
- #define ICP_HSSVOICEDRV_CHAN_BYPASS_DISABLE - ioctl command to remove a unidirectional channel bypass between the channels specified in the data structure of type icp_hssvoicedrv_channelbypass_s passed as parameter.
- ICP_HSSVOICEDRV_STATS - ioctl command to display the stats for the HSS Voice Driver.
- Read – read data on an active channel
- Write – write data to an active channel
- Close – frees up the device driver.

For the remainder of this document we use abbreviated terms for the above functions as follows:

Open, PortUp, PortDown, ChannelAdd, ChannelRemove, ChannelUp, ChannelDown, Read, Write, Close

The mapping of the abbreviated term to real function should be obvious to the reader.

The parameters to each IOCTL are explained in further detail in API documentation in Appendix 0. There are certain options within the API that place limitations on the system, such that if those options are chosen it is not possible test all product requirements under certain configurations, these limitations are as follows:

- Internal loopback of the HSS Voice can only be configured with analog mezzanine configuration in the PortUp command. This limits channels to 32 per port. It is not possible to test the requirement of 128 channels using this configuration because 3 HSS ports @ 32 channels each only allows for 96 channels.
- Internal loopback configuration will not work if there is an external mezzanine plugged into the same port being configured for internal loopback. This makes it impossible to create an automated test run to test all possible combinations on one system
- Analog mezzanine does not provide a loopback mode so is not amenable to fast test execution tests which look at lots minor variations in channel configuration (The external test equipment is ~ 100 times slower to configure and run and get results from)
- byte swapping cannot be tested in internal loopback or framer loopback , this is due to the asynchronous nature of framer loopback it is not always possible to read back data as it was written in the same order
- External test equipment is not able to test bit endianess or bit inversion or Idle Modes

### 2.1.2 Test Code Design Implications

This thesis compares the testing techniques of the current test suite based on the requirements driven methodology of test case selection against the methodologies of boundary value analysis (BVA) and Defined-Used (DU) testing applied in a black box scenario. In order to complete the analysis of testing techniques within a reasonable timeframe the comparison of test methodologies has been narrowed down to apply to the HSS Voice Driver only.

As described in the previous section, the HSS Voice driver provides functions for configuration of the driver and for transmitting and receiving of data. The tests for this driver need to ensure that the driver can be configure the system successfully, transmit and receive traffic, and are able to free resources afterwards.

The T1E1 Framer is used as the external interface to loopback traffic generated internally by the test code. The T1E1 framer configuration is common across all 3 techniques with the exception that the requirements driven test suite uses internal and external traffic generation. Voice traffic is required in order to verify that the system setup is functioning as expected. Voice traffic can be either generated internally via the test code or externally using voice traffic generator equipment. The major differences in external and internal traffic generation and its implications to the test environment are explained below.

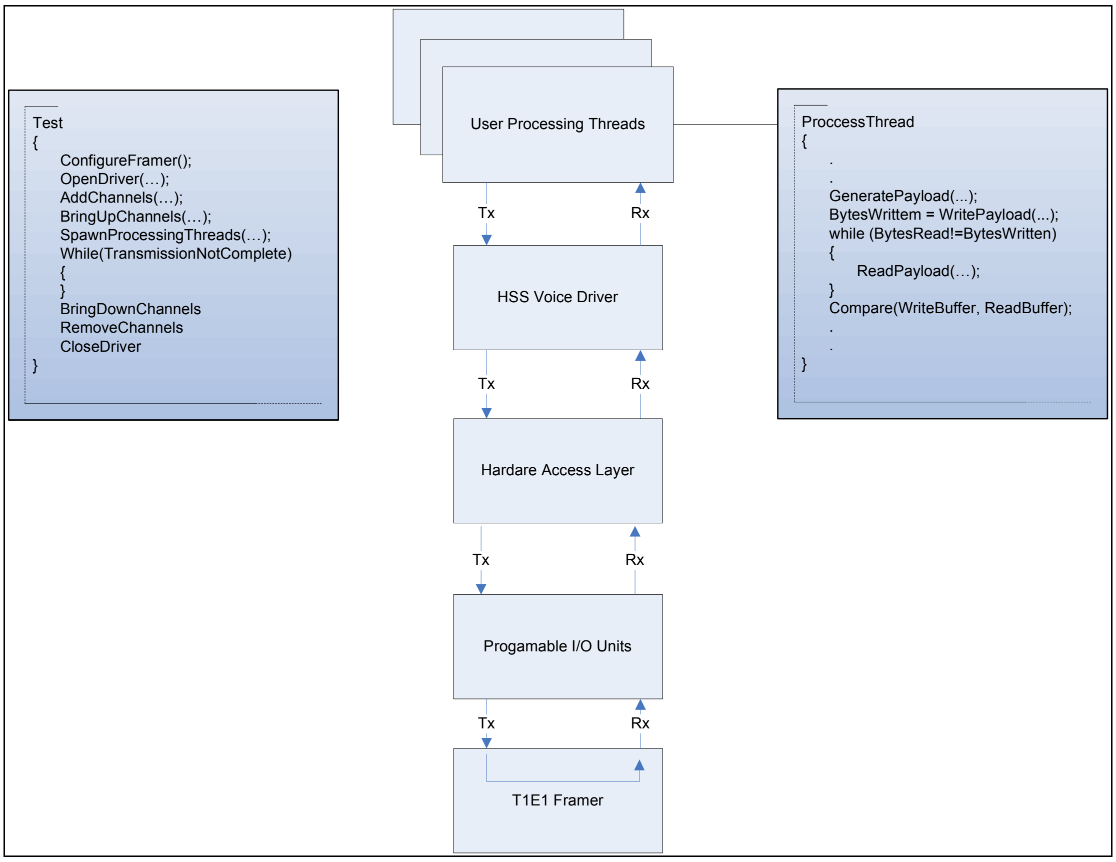
#### 2.1.2.1 Internally generated traffic

Internally generated traffic can be created for every byte in every channel. Every channel requires a 16 bit channel number identifier, a 16 bit payload length and a payload length of 80, 160, 240 or 320 bytes. The payload can be generated using rand (), a random number generator function, provided by the C stdlib, and the seed value can be generated using system time. Multiple channel data can be grouped together in one data buffer and written on one file descriptor, as illustrated below:

Data Buffer: len | channelId | payload || len | channelId | payload || etc.

The Hardware Access layer software copies this data to its own internally allocated memory then passes this data on to the Programmable I/O unit (PIO). The PIO processes the data for transmission on the HSS Bus. The T1E1 Framer device receives this data on its Tx line and, when set to internal loopback, it loops the data back to the Rx line to be sent back to the Programmable I/O unit. The Programmable I/O unit places the received data into a memory location set by the Hardware Access layer driver, which in turn copies the data to the user space application. The programmable I/O unit transmits idle data when there is nothing to be sent from the Hardware Access layer driver. This means that idle data is looped back and is piped through the system to the user space application. The implication of the transmitted idle bytes from the Programmable I/O unit means that the received payload in the user space application payload could also contain all idle bytes, a mix of idle bytes and bytes transmitted by the user, or all the bytes transmitted by the user. It could take more than 1 read of the HSS Voice driver to retrieve back all the original data. To be able to filter out idle bytes, the internal generation of payload data above must ensure that no idle byte value is inserted into the payload. Once the receive side has read back the number of bytes transmitted the test code performs a compare of the data written and data read. This is illustrated in Figure 2 below. The transmission is successful only when the data matches exactly, if not, the sent and received data is printed out for the tester to analyse.

The drawback of this type of testing is there is no measure of time taken to send and receive back the data. Minimisation of transmission delays is important for the end user experience in voice transmission systems. To measure the transmission delay this way is a catch22 scenario. The test application is using up CPU resources in processing the received data, which in turns adds delays to the processing done by the Hardware Access layer. To measure transmission delays of the HSS Voice driver and dependent software requires the use of externally generated voice traffic and minimal user space processing.



```
Test
{
    ConfigureFramer();
    OpenDriver(…);
    AddChannels(…);
    BringUpChannels(…);
    SpawnProcessingThreads(…);
    While(TransmissionNotComplete)
    {
    }
    BringDownChannels
    RemoveChannels
    CloseDriver
}
```

```
ProccessThread
{
    .
    .
    GeneratePayload(...);
    BytesWrittem = WritePayload(...);
    while (BytesRead!=BytesWritten)
    {
        ReadPayload(…);
    }
    Compare(WriteBuffer, ReadBuffer);
    .
    .
}
```

**Figure 2 - Test Code Design for Internally Generated Voice Traffic**

### 2.1.2.2 Externally Generated Traffic

Externally Generated Traffic can be done using off the shelf voice traffic generators. These provide the ability to:

- Simulate voice traffic over various interfaces (such as T1E1, Analog and VoIP),
- measure the transmitted and received signal (from the DUT) , analyse it, and provide a Voice Quality Score such as PESQ score,
- Measure the delay in between transmitted and received signal,
- Many other measurements, echo, signal loss, ect….

The test code requirements for this are simple: Use the driver to configure the system to match the external traffic generator, i.e. match the timeslot on the E1 line. Then setup threads to read the HSS Voice driver and write back the received data to the HSS Voice driver. The voice traffic generator does the rest of the work. The cons to this method of testing are that; it generally not possible to automate the configuration of the test equipment to match the configuration of the test. Also whilst it is possible to automate the loading and running of a manually created configuration, it still takes time to configure the voice traffic generator for each test, and this results in longer test execution times. There are no external voice generators available to test multiple timeslot channels, which is part of the HSS Voice driver API.

### 2.1.3 The Test Environment for Test Method Comparison

The Internal loopback traffic scenario is able to test all the HSS Voice driver configurations, with the only drawback being that it cannot accurately measure transmission delays in the system. The test selection methods being investigated are functional black box testing methods, where as transmission delay is more of a performance issue, which is considered outside the scope of this investigation. Based on this test cases have been developed using BVA and DU test cases to test in an internal loopback environment only.

The existing requirements driven test cases use a combination of externally generated traffic and internal loopback tests. The amount of tests using the T1E1 framer configured for E1 is rather limited due to the one timeslot limitation of the E1 voice traffic generator equipment. The Analog mezzanine is used far more frequently in tests because the Analog mezzanine is able to sample at variable rates and use one or more HSS timeslots accordingly. For comparison purposes these tests will be run as a means of testing timeslot configurations, there are E1 tests that test all 128 channels and Analog tests that test use of multiple timeslots, and for this it should not matter if the traffic is internally or externally generated.

For E1 internal loopback, data can be transmitted and looped back on a complete digital transmission medium, no data loss should occur, making it possible to do a direct comparison of data transmitted vs. data received which is looped back by the T1E1 framer device  The decision to use the T1E1 interface is based solely on the fact that it can provide the maximum number of channels (128) that the HSS Voice driver supports, where as the Analog Mezzanines can only provide a maximum of 12 channels. The comparison of methods has been applied on the use of the HSS Voice driver only.

## 2.2 Version Control

The company uses an off the shelf product by IBM called Rational Clearcase as means of version control on source code. From this it is possible to extract versions of code based on a date or on a release that is labelled in the version control system.

A release package is created using scripts which extracts code from the version control system. The release package is a zip file that contains all source code and makefiles that is able to be compiled on a Linux system. In this thesis we report on the testing of an internal build55 and version 1.0 which was released to the customers.

## 2.3 Software Engineering Practices

Software development within the EP80579 project uses the waterfall lifecycle model for development and maintenance activities. The model is used on a per release basis. The phase for design, coding, testing and release are clearly defined at the projects conception. A preceding phase must be completed before the next starts; phase completion is judged by the outcome of the phase matching the requirements defined by the previous phase. In terms of how this applies to development and test teams at the company; Software development Engineers review product requirements with marketing and agree on what can be delivered and when, normally this is in a series of phased release with incremental functionality added with each release. Each release then has its own cycle of design-code-test-release and only when this phase is complete does the process move onto the next phased with new features.

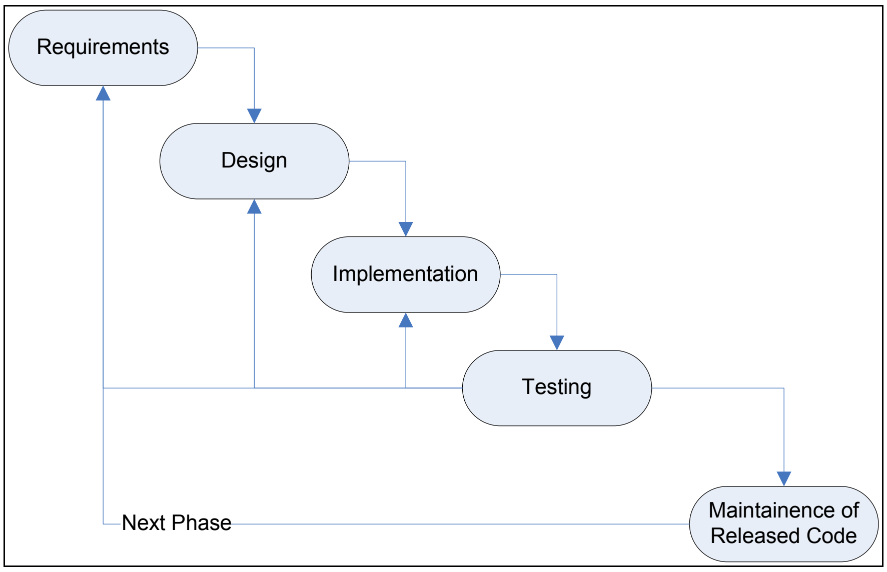


**Figure 3 – Waterfall Lifecycle Development Model**

One of the weaknesses of this model is that requirements change during the design, coding and testing phases. At the company this is handling by a process of change management. Change requests are analysed by development and test engineers. An impact analysis is performed and if the impact is acceptable in terms of value added vs. delay of product then the appropriate changes are added to the project.

Software development engineers write the design documents that aim in helping coders implement code to meet the software requirements. The system test engineers are involved in the review of these documents. This helps to identify at an early stage that the proposed design will cover the software requirements. A poorly conceived software API that requires re-design in the testing phase can cause considerable extra cost to a project, such as the wasted effort in coding the initial API + the test code that went with it.

The test engineers use the design documents and software requirements to put together a list of tests that will validate the software requirements. The development team are involved in the review of test specifications to ensure that the testing covers all the software requirements.

Whilst the developers code up the software the system test engineers write their test applications so that once the developers are complete the test team is also ready to run their test applications to validate the requirements. This approach minimises delays in making a release to customers. The development engineers are responsible for unit testing and integration testing of their own code. Unit testing is aimed at testing the API of one component, checking boundary, return values and out of order calls to the API. Integration testing aims to run an API with its dependent underlying API's to ensure that components work together. The test engineers perform the system level validation testing. Quite often this can be very similar to integration testing, however, the test team are some what removed from the development team, and their tests are developed based on a specification (via design documents and API's) and the result of this is that the suite of tests developed by the test team can be very different from the way the developers have tested it. In addition to this, the test team try to exercise the system via external means as much as possible. If a system is intended to process incoming data from external systems, this means the test team will use external traffic generators where possible. Whilst the development are more likely to use input test vectors to simulate traffic. The approach of developing tests in isolation from the development team and use of external system stimulation by the system test Team has been effective in finding defects in software that has complete and passed all development integration tests.

The system test team use Clear Quest Test Manager [23] (CQTM), BIRT (Business Intelligence and Reporting Tools [22])reports and in house solution of extracting tests from CQTM and running them on the system under test. This allows for complete automation the execution and reporting of tests results. CQTM is a tool developed by IBM, to manage test cases from conception through to execution. It is essentially a front end to a clearquest database that allows users to set up their own schema, which for the test team at Intel, this allowed for tests to be added to a database. The tests could be

- Arranged into test suites,
- Run in iterative stages,
- Have execution states such as planned schedule, running complete,
- Have result states such as pass and fail.

This allows for live updates of test execution status for long running test cycles, for test suites in the order of 1000-2000 tests that could take up to 2 weeks to execute. Software requirements can also be added to the database, this then allows cross referencing of tests that cover requirements. The tool also allows scheduling of tests and is used in conjunction with an in house developed test Management system, the test Management system extracts scheduled test from a test suite and runs them on the system under test.

The status of execution can be easily reported to management, via the use of BIRT reports, on how many tests have been competed, how many have passed and how many left to run. BIRT reports is an Eclipse based tool that allows report generation from databases.

# 3 Related Research Work

In this section we discuss some of the work done by others in the area. We focus on Black box testing methods and automation. The reason for investigating black box testing is that white box testing is considered costly/time consuming to develop test for. We want to find if there are more efficient methods for black box test generation. We also look at research done into automatically generating test cases as this also has potential to save time in test development.

There have been many investigations done on comparison of black box testing techniques, measuring the effectiveness of test techniques and, automation of test generation. This section summarises some of the previous work that has been done in these areas that is of relevance to this thesis. A common theme that seems to come out of the comparison of methods is that it is better to apply a combination of test selection methods then using one alone. Each methods success seems to vary depending on the program and normally a second method will pick up the defaults that the first method misses. The trade of is that multiple methods takes more time to implement so automation of test case generation is useful to negate this negative of applying a combination of methods.

## 3.1 Comparison of Testing Methodologies

There has been previous work done on the comparison of Black Box testing techniques. Of particular relevance to this thesis is the work done by Wood, Roper & Brooks [16], where they look at one of the techniques compared in this thesis, boundary value analysis. Their paper also looks at test generation by code reading by stepwise abstraction, b) functional testing using equivalence partitioning and boundary value analysis, and c) structural testing using branch coverage. Each methods success varied between programs and each method has its strengths and weaknesses. One of the main conclusions was that a combination of testing methodologies was better then any one method used in isolation

Seo & Choi [7] have written a paper on the comparison of five black box testing techniques. Their techniques are use-case driven testing, black box testing using collaboration diagrams, testing using extended use-cases, testing using formal specifications (OCL or Object-Z). They applied the methods to two software systems, one for controlling an ATM and the other was a session scheduling system. They describe the application of each test methodology and list code coverage achieved for each technique as their metric for each method, although it does not specify whether it is line coverage, branch coverage or DU coverage that they are comparing. They do not mention anything about the number of defects found, their summary is more of a recommendation to test planners to what method to choose if they are looking for high code coverage of interface coverage between components of the system. This analysis was not suitable to what the Intel Shannon test team was looking for, as mentioned in previous sections we wanted to look at other methods to see if they are better at finding defects.

Bertolino [10] summarised many of the research articles available today on comparison of software testing and the effectiveness of the various techniques. She states that there are now so many varied methods of test selection it is difficult to justify which one to choose. However it does seem from the work that she carried out that it seems a more effective method testing would be to apply a combination of techniques as each method would be more likely to find a certain class of defects. Interestingly, she makes the following statement: "*Demonstrating effectiveness of testing techniques" was in fact identified as a fundamental research challenge in FOSE2000* Bertolino [10], which shows that there is far more work to be done in this area. More of the recent research has been looking at model based testing, where test are derived of a model of the software such as UML. Such models also are considered suitable to automated generation of test inputs. This report seems relevant to this thesis for two reasons:

1. We are investigating the effectiveness of test selection methods and comparing them.
2. We are also looking to at the automation of test generation

Wegner and Grohtmann [11] have investigated and compared random testing against a technique known as evolutionary testing. Their results showed that for real time embedded systems, it is a far superior method than random test selection. *"When evolutionary algorithms are used to solve optimization problems, good results are* *obtained surprisingly quickly"*. This method does appear very applicable to the system test team. The evolutionary test method is a specialised technique for finding timing related issues. This reinforces the notion by Bertolino [10] that test methods are good for finding classes of faults, in this instance; timing related faults, and should be used in parallel with other test selection methods. This work is well worth further investigation by test engineers of the company.

## 3.2 Software Testing Effectiveness

This section discusses previous work done on the effectiveness of software testing.

In a paper by Frankls [2] he suggests that there is no definitive evidence to prove that any software testing method is effective. The paper investigated the application of the random test case selection on a piece of software approximately 10,000 lines of C code. The random test cases were broken up into suites of tests to cover functional areas of the software. On each test suite he looked at the code coverage that a set of the tests achieved and his summary of this was that the more lines of code covered by tests the more likely to find faults within a system.

In another paper by Chen, Kuo & Merkel[4] they investigate and compare the effectiveness of two software testing methodologies', random test selection and Adaptive Random test selection (see Chen, Leung & Mak [5]). They look at measures of effectiveness (E) – Expected number of failure, (P) – Probability of detecting at least one failure and (F) – the number of test required to run before finding the first defect. Their findings was the based on the F-measure, Adaptive random testing was more effective at finding defects that pure random testing. However they also state that to compare testing effectiveness a high sampling size (test execution) is required in order to be statistically accurate.

Huber [6] suggests a method of process metrics in determining the effectiveness of software testing. He does not specify any particular method, process metrics set testing goals for the test team which act as a motivator to finding tests. For example one metric might be "Find 20% more defects than the last project", "Improve test Coverage by 10%". To do this testing teams need to be innovative and forward thinking. Simply repeating the process used in a previous project would not yield improvements. There is no value added if testers implement 10% more tests but find no extra defects, yet if they write 10% more tests and find 20% more defects then it is quite obvious to management that the work the test team is doing is effective and is improving the software quality. In this investigation we can apply this by recording the time taken to implement the three test selection techniques and measure their effectiveness by the ratio of defects found vs. time taken to implement the tests.

## 3.3 Test Automation

Rajappa, Biradar, Panda [12] have presented a method using graph theory which allows for the automatic generation of test data. This method involved the representation of a directed graph, as an N x N matrix where N = the number of nodes in the graph and an edge between nodes is represented as a binary one and no edge represented by a binary zero. They use a Genetic Algorithm in order to access combinations of test for suitability. The result is the generation of a large number of test cases and high code coverage is achieved. The author feels that generation of tests using such a method is mathematically complex, and the paper also reports that it can lead to regression testing issues. Since the method generates so many test cases it is difficult to select a subset as a regression test suite. It can be difficult to find testers with a high level of mathematical understanding to be able to replicate such a method. With a large number of automatically generated test case it would be difficult to justify test case elimination if were required in order to reduce test cycle times.

Javed, Strooper & Watson [13] have published a paper that reports on the automated generation of test cases. As per other publications discusses here, they mention that model based test selection techniques have become quite popular in recent times and that model based software specifications are easy to automate the generation of test cases. Their particular paper focuses on the automatic generation of test cases using UML sequence diagrams to generate unit test cases. This method appears particularly useful in generating test cases early in the development cycle. In this thesis we investigate a similar method that automatically generates system test data from a RESOLVE like software model.

B.-Y.Tsai, S. Stobart and N.Parrington [14] present an interesting paper on data flow modelling. There selection of test cases is based on a DU pairing of hidden data members in a class i.e. local function variables and private data members. When we compare what they have done it highlights that the DU method applied in this investigation is really state based testing. The HSS Voice driver provides that puts the system into a certain state for each Voice channel in the system. We can verify if a port is up by adding a channel, we can verify if a channel is added by bringing it up, we can verify if a channel is up by transmitting and receiving traffic on it etc. The DU method we have looked at in this thesis is based on a functional specification of the API; With the DU analysis, we can tell the state that the driver, or channels on the driver, are in. Internal structures of the HSS Voice driver that do not define the state will be ignored by the DU method used here because we have approached the system as a black box and are not aware of the internal implementation. One of B.-Y.Tsai, S. Stobart and N.Parrington [14] main points is that data flow testing can be used in conjunction with state based testing and that the two methods will find different classes of faults. The remainder of their paper focuses on the data flow testing as a white box testing technique which is outside the scope of this investigation.

The next paper we discuss is the motivator for our 3rd method of test selection in this thesis. Edwards [15] discusses a software specification called RESOLVE (see Sitaraman M, Weide [18]) that can be used in conjunction with flow graphs to generate black box test cases. Edwards [15] states the following about the RESOLVE specification: "In such a specification, an abstract mathematical model of client-visible state is associated with each type or class, and each operation or method is characterized by pre- and post conditions". With this information it is possible to automatically generate state based tests based on the black box specification of the object under test.

Edwards [15] references use a flow graph in order to help define test cases. The flow graph defines functions as nodes and flow of control from one operation to another as edges in the flow graph. Using this flow graph "define" and "use" scenarios can be created. "Definitions" are at a node where the operation can potentially change its value, "Use" occur at nodes were the inputs may affect the behaviour of the operation. Using the RESOLVE specification: Definitions can be determined from the specification. The specification gives an indication of the class members whilst the post conditions of an operation indicate the redefinition of class variables. "Uses" can be determined from the preconditions of an operation. Given such a specification is possible to apply the analogues of white box testing such as DU testing to apply in a black box testing environment. The flow graph is used to determine valid DU pairs extracted from a RESOLVE specification. Edwards [15] paper uses the flow graph and RESOLVE specification to generate test cases for “all definitions”, “all uses” and “all nodes”.

# 4 Current System Test Methodology

The method of test case selection used by the test team is based on a requirement driven method of selection. It is adhoc in that test cases are picked mainly to cover product requirements and to test the stability of the software. The software produced is generally a configuration driver that sets up firmware on the system to handle incoming traffic in a certain way. Examples of this include:

- setting up varying cryptographic algorithms of encryption/decryption data incoming/outgoing data streams
- configuring TDM lines to process input on a certain number of timeslots within a TDM T1 or E1 line

These configuration drivers lend themselves well to applying boundary values as test input parameters. For example "*test one timeslot channel on a TDM line then test all 32 timeslots of an E1 TDM line assume that all variations in between are ok*". Otherwise test selection is based on software requirements and ensuring that a test exists that covers that requirement. The problem with this is that while it is ok for validating requirements it is not targeted to finding defects. In addition to this there is a considerable effort put into implementing and executing test for a feature.

Development teams use the same methodology as the test team for selection of test cases and input data. The result is that two different teams work independently in parallel to develop tests that could end up being very similar in nature. The test team execute their tests after development, whilst the test team invariably find defects; there is a likely hood that the amount of effort put into implementing test code is yielding limited results. Is there a method of test selection that is either better at finding defects, quicker to implement and execute or both?

## *4.1 Alternative System Test Techniques Investigated*

In this investigation we compare the existing requirements driven test selection method to other black box test selection methods. There are several methods to choose from, the methods listed in section 1.3.4 list the most basic and well known methods. There are many more methods of a far more complex nature based on mathematical models, structural models and other models. Some of these have been mentioned in section 3. For this investigation we wanted to take the learning's from the modules taken in the MSc Software Engineering course and take something from the outside software testing community. We chose the boundary value analysis as one of the Black Box testing methods that was covered in the MSc Software Engineering modules. It is a method that finds a well known source of faults in many software projects, it is easy to implement and does not require experienced software engineers to use it. We reviewed several papers for an alternative Black Box testing method. Some of these have been listed in section 3 of this thesis. For our second method we chose the DU test cases generated from the RESOLVE method as described in Edwards [15] and discussed in section 3.3. Other method were considered and have been have discussed below.

### 4.1.1 Requirement-Based Automated Black-Box Test Generation

We looked at a paper from Tahat, Bader, Vaysburg & Korel [19], who investigated, requirement based automated black box test generation. Their report discussed automated generation of requirements from specification description language (SDL). SDL is claimed to be highly suitable to real time and embedded systems. SDL is well suited to full scale projects because of its abilities to interface with other languages. Such languages include other high level notations for analysis such as unified modelling (UML). Furthermore there are tools available that can generate executable code such as C\C++. It was felt by the author that the time taken to become familiar with the technique does not fit into the timeline for submission of the thesis. Also the requirements of the HSS Voice driver are quite vague and it was felt by the author that an accurate SDL model could not be constructed to create an auto generated test suite. We also wanted to look outside requirement driven test selection as we are already using this method very similar (although not based on SDL).

### 4.1.2 Structurally Guided Black Box Testing

Kantamneni, Pillai, & Malaiya [20] have published a paper which describes a method of structurally testing a system using Black Box techniques. Structural testing involves deep inspection of the code and the creation of tests to cover all parts of the code. Common Coverage criterion includes branch coverage and statement coverage.

Techniques normally used to creates test for these are path testing, DU testing, all use testing etc. This is the domain of white box testing as it is not possible to see the internals of the code to be able to structurally test it in Black box testing. How can structural analysis be applied to Black box testing? Also how can a reasonable amount of code coverage using traditional black box be methods be achieved? Kantamneni, Pillai, & Malaiya [20] approach is to develop a set of tests, analyse the coverage, and then develop more tests to try to get 100% coverage. This approaches does not lend it self to automation, although the authors of the paper have suggested that their method will allow auto generated test cases, perhaps this is suitable to certain systems. It does not seem suitable for the system we are looking at. Whilst there is a tool called gcov for measuring code coverage on Linux systems it is not known if this will work on the Linux 2.6.18 kernel, it was originally developed for the 2.4 kernel and there are some reports that there are patches for the 2.6 Linux kernel however there is no guarantee that if we chose to explore this method further there we would be no guarantee that we would be able to measure code coverage using any of our test methods.

### 4.1.3 State Based Black Box Testing

State based black box testing, views the software under test as a series of states and transitions between these states and the inputs and events that cause these states. Test cases are generated that exercise cause changes in state. The tester is required to verify that events and inputs change the state as expected and are also required to monitor for any other activities that occur due to a change of state. A state transition graph may help identify unreachable states or dead states that cannot be exited. Because of State based testing being a typically black box style of testing, the actual internal state of the system under test is not easily visible. Because of this, the problems of Control and Observation come into play. Control is the ability to ensure that the correct start state for a test exists. Observation refers to the ability to see the final state of the system after a test has been run.

State based testing allows for easy automation of test case generation, there must be an explicit mapping between the elements of the state machine (states, events, actions, transitions, guards) and the elements of the implementation (e.g., structures, functions, parameters etc.)

The following Checklist for analysing that the state machine is complete and consistent enough for model or implementation testing:

- one state is designated as the initial state with outgoing transitions
- at least one state is designated as a final state with only incoming transitions; if not, the conditions for termination shall be made explicit
- there are no equivalent states (states for which all possible outbound event sequences result in identical action sequences)
- every state is reachable from the initial state
- at least one final state is reachable from all the other states
- every defined event and action appears in at least one transition (or state)
- except for the initial and final states, every state has at least one incoming transition and at least one outgoing transition
- for deterministic machines, the events accepted in a particular state are unique or differentiated by mutually exclusive guard expressions
- the state machine is completely specified: every state/event pair has at least one transition, resulting in a defined state; or there is an explicit specification of an error-handling or exception-handling mechanism for events that are implicitly rejected (with no specified transition)
- the entire range of truth values (true, false) must be covered by the guard expressions associated with the same event accepted in a particular state
- the evaluation of a guard expression does not produce any side effects in the implementation under test
- no action produces side effects that would corrupt or change the resultant state associated with that action
- a timeout interval (with a recovery mechanism) is specified for each state
- state, event and action names are unambiguous and meaningful in the context of the application

# 5 Test Case Selection Techniques

This section describes the decisions that were made in how selection of test inputs are handled for each test method

## 5.1 Boundary Value Analysis

Boundary value analysis technique was chosen as a test technique as it is easy to apply and it focuses on finding a common source or faults which is at the boundary of inputs values to an API. The boundary value analysis tests exercise each input of the API as listed in Appendix 0. There are some functions in this API that don't have explicit input values however the description of the API and the design document indicates the presence of certain internal counters that puts range limits of functions. The inputs have also been selected based on the policy of positive tests only. This is a policy decision that is based on the assumption that the customers of the product have expert software engineers that use the API to create their own programs and should be able to use the API as intended. Based on this the inputs and boundaries have been identified as follows:

Open can be called to open the driver up to 128 times. This indicates the presence of an internal counter required to check the open limit. Therefore the boundaries identified for open are 1 to 128

PortUp requires a HSS port identifier, a port configuration and a loopback mode. There are three HSS ports on the system under test, each value is expected to be processed the same regardless of its value, therefore the range is 0 – 2. The port configuration for our BVA tests will be hard coded to use the T1E1 framer, there is an internal loopback configuration, however we chose to not to test this option due to the limitations on automation (see section 2.1.1 for more details on configuration limitations). There are 2 other configurations that are not possible to test because the T1E1 framer hardware does not support it (They are there to support a customer who provides their own T1E1 framer)

ChannelAdd requires: a channel number from 1-128, a HSS port from 0 - 2 to link the channel to, a HSS port timeslot map to assign the channel too.

The API description mentions that up to 128 channels can be added to a client, so there must be in internal mechanism that tracks this. A client could potentially add all timeslots as well so we need a range for this. We should test 1 channel on a client and 128 channels on a client and we should test 1 timeslot used on a client and 128 timeslots used by separate clients.

The remaining functions do not require boundary value analysis due a  policy of positive tests only. This policy is based on the assumption that customers will use the software with the correct inputs, so invalid inputs do not require testing. Secondly the boundary value analysis was intended to tie into the existing test code framework which has not been designed for negative testing. The only option after channelAdd is to:

- bring all the added channels up,
- write and read on all the added channels
- bring all the channels down
- remove all channels
- close all clients using /dev/hss_voice

The ranges for each function have been detailed in Appendix II

## 5.2 DU Coverage Applied to Black Box Testing

This section describes the RESOLVE like model used and how test cases can be extracted out of it. The main reason this method was chosen was that it allows for automatic generation of test cases. Automatic test case generation makes our test development phase more time efficient, later on, in section 9 of this document, we discuss the results of using the RESOLVE like model in automating the generation of test cases.

The RESOLVE discipline as described by Sitaraman & Weide [18] is quite complex. It requires working with experienced engineers who have used the discipline to gain a full understanding on how to define it correctly. Likewise, the RESOLVE specification was not design for test automation either - It is a formal specification for a software design. In this investigation we used the RESOLVE principles described by Edwards [15] to create DU test cases based on a function description. We define a RESOLVE 'like' specification which can be used to extract definitions and uses of variables that are described in the API documentation and design documents to create DU test cases in an attempt to structurally test the HSS Voice Driver.

The RESOLVE 'like' specification contains global variables; the global variables are equivalent to the global context in the true RESOLVE specification. It contains an interface that defines the API operations; each operation uses the keywords from RESOLVE that have been interpreted in Table 1 below:

| RESOLVE operation keywords | Usage |
|---|---|
| Alters | the functions alters a global or internal variable |
| Consumes | these are parameters to one function of the API |
| Produces | output or return of the functions |
| Requires | use case of a global or parameter to a function |
| Ensures | defines or redefines a global or local variable of a function |
| Preserves | value is constant in this function (not uses in our analysis) |

**Table 1 - Resolve Keyword Mappings**

For our testing purposes can use the following keywords to extract DU pairs:

- Global variables and ensures are the instances of D and,
- Requires are instances of U.

Edwards [15] uses a flowchart in his method to help select valid paths, for the HSS Voice Driver this is not required. The normal order of calls to the HSS Voice driver is as follows:

1. Open

2. PortUp
3. ChannelAdd
4. ChannelUp
5. Read/Write in any order
6. ChannelDown
7. ChannelRemove
8. PortDown
9. Close

Valid DU test cases can be determined from the operations of the RESOLVE 'like' specification that specifies its operations in the above listed order, provided that the following rules are adhered to:

- Ensures are defined do not use AND or OR
- Requires use AND or OR for conditional statements
- If a variable being analysed for its D and U cases is not redefined before it is used then it is a valid test case. If a function is called in the above order before a use case then we skip calling that function.
- If a variable is defined below a use case in the specification we have to call the functions in the above listed order again skipping any functions that re-define until we hit the use case
- Close cannot be called without a valid file descriptor, an invalid file descriptor does not test the HSS Voice driver at all as a valid file descriptor is the only link to the HSS Voice driver

Due to the fact that we want to exercise as many DU combinations as possible, we end up creating test cases that require incorrect use of the API in order to follow a code path that exercises the DU pair. The company policy as described in section 5.1 is not to implement these types of test cases. However with this method of test case generation we get these tests for free, and if we did not execute these then the method essentially is reduced to an equivalence partition method that is equivalent or less likely to detect defects than Boundary Value Analysis. We can check that it is not possible to add channels to a port that has not been configured, we can check if we can re-configure a port, try to bring up a channel that has not been added, this method will check a whole range of these types of issues. The RESOLVE ‘like’ specification that we have derived for this investigation is listed in Appendix III.

## *5.3 Requirements driven Selection*

This section briefly describes the scope of testing of the EP80579 IP Telephony Package as defined by the system test plan. The system test plan is a required document as part of the product development process that describes what is to be tested by the system test team and how many resources are required to execute the plan. This section looks in more detail at the potential overlap with development integration testing.

Figure 4 shows components of interested to the system test Team. The lightly shaded components have been planned to be tested by system. The development team test the lighter shaded and darker shaded components. Each team produces their own implementation of test code.

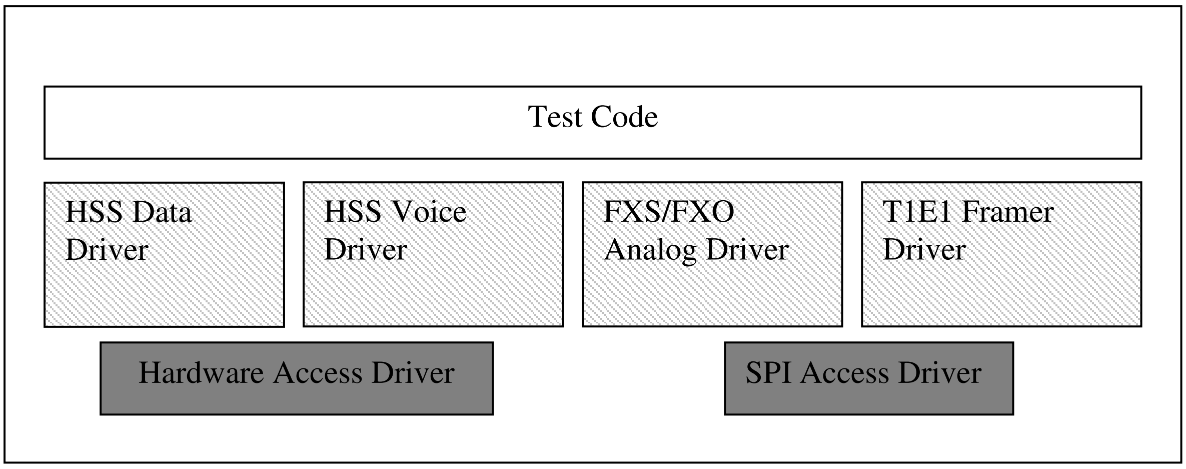


**Figure 4 Software Components to be Tested**

To help minimize overlap between system test and development team and maximize the system test team effort to find customer scenario defects. API verification is outside the scope of system test and is regarded as a unit test activity; therefore system test code policy is not to check the following:

- Parameter checks

- Boundary conditions for API parameters
- Return codes
- Out of sequence API calls

The plan by the system test team to test the HSS Voice driver is to use the API to setup channels and the Linux user space read/write access to create or loopback traffic to external sources. Tests are selected to cover each requirement of the product requirements document and if specific internals in suggested by high level document then tests are also selected to target such internal mechanisms. Based on this, the tests selected have been listed in Appendix IV. There are a total of thirty nine tests that have implemented to test the product requirements.

These tests cover much more than the BVA and DU method test case. To implement tests within the timelines of the Thesis, the scope has been cut down to only test the HSS Voice Driver using the T1E1 framer as the external interface. The tests listed in Appendix IV test:

1. combinations of the all drivers
2. combinations of channel sizes
3. all Analog interfaces
4. all T1E1 interfaces
5. specific firmware processing features – bit endianess, bit inversion
6. design features – handling of transmit over flow scenarios
7. stress testing – continually enabling and disabling channels whilst data is processed on other channels

Since BVA and DU tests do not cover items 1, 3, 5, 6 these tests should be excluded from the comparison. For item 3 however it is normally possible to adjust these tests to use the T1E1 interface. They use a combination of channels sizes which is not possible to test using the external test equipment and due to test code limitations is not possible to test in an Internal Framer loopback scenario. Therefore all tests using the analog interface shall be included. The author is confident that he can filter out defects that are specifically related to the Analog Driver in this instance. This shall be discussed further in section 8.

# 6 Generation of Test Cases

This section describes the generation of test cases for the three methods being compared. As a general rule the policy of the system test team is to test positive test cases only. For the following reasons:

1. The role of testing out of order API calls, boundary value, and parameter checking has traditionally been the responsibility of the development teams in either their unit or integration tests. If development have implemented their tests correctly there should be a low probability of finding defects in negative test cases.
2. Intel customers will have their own experienced software engineers developing a product to be sold onto the market place. These customers are considered quite capable to use the API correctly. Intel also provides Engineering support to help customers iron out any usage issues.

The risk of the system test team not performing negative test cases has been ranked as low based on the above to factors. Therefore this strategy will also be applied to BVA and the DU test case generation.

## *6.1 Current Test Framework*

The system test team at the company have a framework for testing that is re-used from project to project. This framework allows for the storage of test inputs in a C code file and accompanying files that contains functions to access the test inputs to configure the system with. The framework is dependent on a testCli application which allows the user to invoke functions from a loaded kernel module (Linux kernel space code) or shared object library (Linux user space code). The test code is compiled into either a shared object library of kernel module, which is then loaded in into the testCli application for execution.

### 6.1.1 Current Test Design

The current test design has been developed to work with the framework described in the previous section. Test input is managed via an array in a C file. Each array index is a struct containing individual test data the array of structures is shown in Figure 7. The structure of this is shown in Figure 5 below

```
typedef struct sysTestHssDrvDesc_s
{
    uint16_t sysTestNum; /*Unique test identifier*/
    uint8_t NumberOfVoiceClients; /*number of file descriptors to
        be opened*/
    uint32_t iaSysTestPortCfg[NUMBER_OF_PORTS];
        /*This array contains the data required to configure
         * the HSS ports*/
    iaSysTestHssDrvChanCfg_t
        iaSysTestDrvChanCfg[NUMBER_OF_PORTS];
        /*pointer to a channel configuration structure, which
        contains the configuration for adding
         * specific voice channel(s)*/
    uint32_t numActiveChans[NUMBER_OF_PORTS];
         /*number of channels to be configured on a port*/
} sysTestHssDrvDesc_t;
```

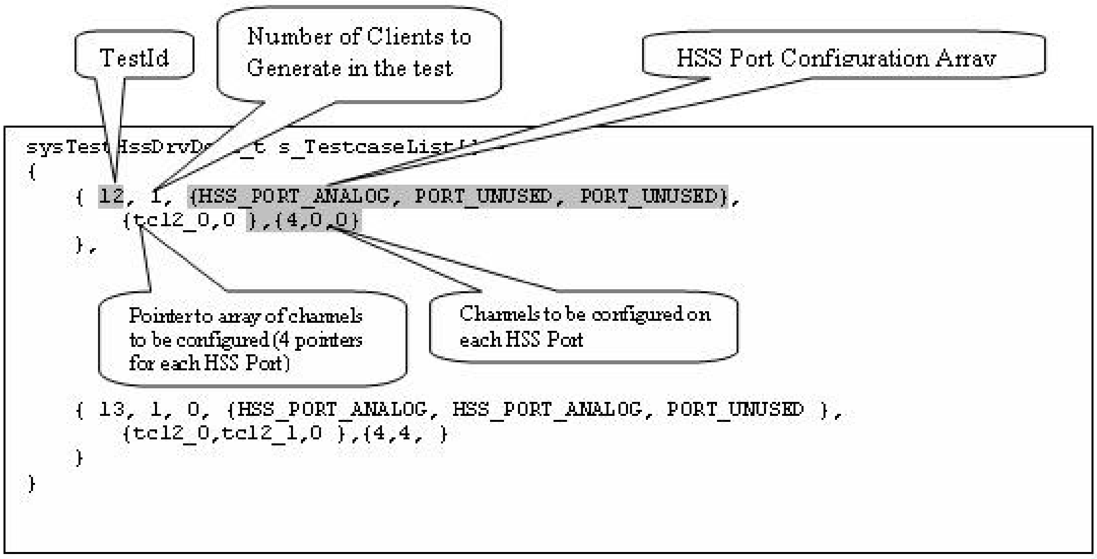
**Figure 5 -** test **List Structure**

The iaSystestHssDrvChanCfg_t is described in Figure 6 below

```
typedef struct iaSysTestHssDrvChanCfg_s
{
   uint32_t clientID; /*this tells the test code which file
        descriptor this channel will belong to*/
   icp_hssdrv_timeslot_map_t tsMap;
       /*Timeslot map is a 128bit struct made up of 4 32bit
        integers and is a bit mask of the timelots to be used on
        this channel.
   sysTestHssDrvChanParams_t params;
        /*parameters specific to this channel, such as channel
        size, bit inversion, byte endianess*/
} iaSysTestHssDrvChanCfg_t;
```

**Figure 6 - Channel Configuration Structure**

The list of tests in the array looks like the following:


TestId
Number of Clients to Generate in the test
HSS Port Configuration Arrav
Pointer to array of channels to be configured (4 pointers for each HSS Port)
Channels to be configured on each HSS Port

```
sysTest HssDrvDe _t s_TestcaseList[
{
    { 12, 1, {HSS_PORT_ANALOG, PORT_UNUSED, PORT_UNUSED},
      {tc12_0,0 },{4,0,0}
    },

    { 13, 1, 0, {HSS_PORT_ANALOG, HSS_PORT_ANALOG, PORT_UNUSED },
      {tc12_0,tc12_1,0 },{4,4, }
    }
}
```

**Figure 7 - Structure of List of Tests**

The test code provides functions to lookup the testId on the list of tests. It then knows which ports to be configured and how they should be configured. In the example shown in Figure 7, test12 uses the 1st HSS port and will be configure for analog. The channel configuration pointer, tc12_0, describes the channels that will be added to the 1st HSS port. There are four channels to be added to the first HSS, this is indicated by the last block of values in the test structure, and this indicates how many channels are described in tc12_0. Test13 uses two HSS ports to be configured for analog, as can be seen in Figure 7; test13 re-uses the same channel configuration tc12_0 and also adds four more channels to the 2nd HSS port with a different configuration. There are many optimizations that could be done to the layout of this code however that is outside the scope of this investigation.

The test code also provides packet creation and transmission functionality, this has already been described in section 2.1.2.1. The existing test code also provides functionality to bring down, remove and close all the channels to return the system to its original state. There are also some other functions provided to re-map timeslots of channels and change the channel parameters as listed in the HSS Voice API in Appendices
HSS Voice API

### 6.1.2 Test Automation

The testCli application is a command line interface that allows for automation. Normally automation is done using Perl scripts with the Perl Expect.pm module. Perl Expect is used to check for return codes or output from functions called from the testCli. Perl can also be used to interface with external test equipment. This system allows a Perl script to first configure the system under test, then configure external test equipment to inject traffic into the system, then check the system under test for its response by verifying that it received and was able to process all the traffic sent by external equipment or whether it is still stable or check the external test equipment for responses from the system under test. For example in the current tests design for the HSS Voice Driver, the Perl script automates test code in the following way:

1. Open the testCli,
2. Load the test code shared object,
3. Call the Run command with the relevant test ID, which configures the channels and kicks of processing on configured channels,
4. Call a system command to tcl which opens a connection to the test equipment, loads a specific traffic configuration file then starts transmitting traffic for a specified time. The test equipment creates a results log file which records average voice quality scores[2] and some other statistics,
5. When the system command returns, it is assumed the test equipment is complete. The script calls a Finish command to remove all channels and close the Voice Driver,
6. The script opens a log file created by the external test equipment and checks that the average voice quality score is above a specified value.

The above example is for externally generated traffic, for internally generated traffic steps 4-6 can be replaced with:

4. start transmitting traffic on all channels

[2] Voice Quality Score is used as a pass/fail criterion for externally generated traffic

5. wait for complete message[3], call Finish command to remove all channels and close driver

There are some variations on the above for specific tests, but otherwise this pseudo code covers the automation on 80% of the tests. The remaining 20% of tests can be difficult or not worth to automating. For example, one of the parameters of the Voice driver is a bit endianess performed on a channel, this is to enable communication between big endian and little endian devices. However it cannot be tested in an internally looped back scenario; as the data has its MSB switch on transmission and switch again on receive by the Programmable I/O unit, but to the user it is not possible to see if this switch ever took place. It cannot be tested by External traffic as it only supports 1 mode of endianess. The only way to test it is to create 2 channels on the same timeslot but on different HSS ports. One channel is big endian and the other little endian; loop a cable from one HSS port back to the second HSS port. The traffic is internally generated from one channel and received on the 2nd and each byte should appear swapped from how it was transmitted. It takes very little to execute this test, but the physical configuration of the cables needs to change each time the test is run, and this makes automation of this test a waste of time. All the others tests can be run with a common configuration

## 6.2 Requirements Driven Test Generation

The test inputs for test code generated at the company are based on ensuring that every requirement has at least one test that covers it. The test inputs are selected on a manual basis, however there is a bias in selecting the minimum and maximum values when the API or requirements provides/requires a maximum number of supported items (In the case of the HSS Voice driver, the API supports a maximum number of 128 channels). The minimum values to an API is always chosen for the reason that system test engineers always need a simple test to debug their own test code before they start running tests that involve large input values that stress the system. Another criterion for test input is to mix up the inputs a bit, if there is an option to run different configurations together. In the HSS Voice driver this meant creating a client that has single, twin and quad timeslot channels, have those channels overlap their timeslots

[3] For Internal loopback traffic matching all packets that were sent is the criterion for pass or fail

and having multiples of single timeslot channels between quad timeslot channels[4]. Finally, if there is any customer use case scenarios provided in the development documentation, then test case are written to cover these scenarios.

For the HSS Voice Driver there are thirty nine test cases specified that test the HSS Voice Driver and all its sub-components together – see Figure 1. To create a comparable environment there are some test cases in this test suite that test features that we have purposefully excluded from the BVA and DU tests as they are not required to test the features that the HSS Voice driver provided these include:

- tests using analog driver and analog mezzanine
- tests using the Hardware Access layer loopback

Some of these tests have been converted to framer loopback where possible. Generally it is was not possible to convert them to use the external traffic generator as they are tests that use multiple timeslots of which the external traffic generator does not support. Of the thirty nine test cases – twelve can be changed to framer loopback with internally generated traffic and four stay as is. This left one test that used the external traffic generator, so it was decided to convert it to framer loopback as well, the consequences of this has already been discussed in section 2.1.2. The list of HSS Voice test cases that were executed, were adjusted and run as is, is listed in Appendix IV.

## *6.3 BVA Test Case Generator*

This section describes the aims and design issues of the BVA test case generator script.

### 6.3.1 Deciding how to generate the test code

In order to create BVA test cases with the least amount of effort it was decided to design a test case generator that creates an array of test inputs that can be executed

[4] Multi timeslot channels do not have to be contiguous, therefore it is possible to interleave timeslots of channels

using the current framework and supporting test API as described in section 6.1.1. Two methods were considered for the test case generator:

1. A C program could open the file and parse the input to create boundary values for each function and create a matrix of tests that exercise the maximum and minimum of each boundary value in at least 1 test or,
2. A Perl script could be written to do the exact same as above

A perl parser was designed to take as input: a RESOLVE like specification to extract the test inputs and generate DU test cases  The parsing of the input file depends heavily on string matching and splitting of strings. The author had more confidence in Perl being able to do this as string parsing is one of Perl's strong points, so Perl was chosen for the task of parsing the input and creating the test input data for the test framework

### 6.3.2 Defining the API Boundaries

It was hoped to parse the RESOLVE like specification list in Appendix III. However it was not immediately obvious what the boundaries are in this specification. Upon closer inspection it can be noted that many of the boundaries for the API are in fact meta data visible to the entire API, and are not explicitly input or output to one particular function. The API functions update this meta data when called; the internals of the HSS Voice driver have boundaries on this internal data. As an example, take the open function: This function can be called successfully up to 128 times, but it does not take in input of 0 to 128. To make these boundaries clear to the generator a far simpler file was created to define these not so visible boundaries, this boundary specification is shown in Appendix II.

As mentioned at the start of this section, the policy of test case generation is to consider positive test case generation only. The implications of this on the BVA test case is that we only have to consider boundaries on: open, portUp and channelAdd. channelUp & channelDown, channelRemove brings up & down and removes the added channels – so the only input is the added channels. portDown brings down the port that was brought up in PortUp so the only input is the port number brought up.

Close can only be called on a valid file descriptor which can only be obtained by calling open in the first place. This greatly simplifies the boundary specifications.

The next issue to consider was the use of bitmasks in the API. The author did not find any publications which gave a suggestion of how to treat bitmasks using Boundary Value Analysis. When using portUp and channelAdd, both require a port number, which applies to a physical port on the system: HSS0, HSS1 and HSS2. If each bit were to result in completely different behaviour such as writing to hardware registers in a driver then obviously each bit should be tested. In our case we are only looking at which port we add our channel too so the behaviour for each bit is the same except for the location of the channel in the system. We chose to use the highest and lowest bit values for these cases.

When using channelAdd, a timeslot bitmask is required, again the BVA method does not specify how this should be treated. The bitmask is a structure of four 32bit integers, this makes 128 possible combinations, which greatly adds to the execution time, but will it find more defects? Again the bit mask does not reflect a difference in behaviour, expect the placement of the channel in the system. It was decided to use the boundaries of each 32 bit integer, so we are testing the lowest and highest timeslot on each E1 line as inputs, this makes reduces the number of combinations to 8.

The resulting boundary definition of the API has been placed in the text file in Appendix II. The definition has following format which the parser expects in order to work properly:

<userInsertedInput> - replace text encapsulated by <..> with your own input such as operations of the API or input and output names of the API
[text] – is optional, but you need to have at least 1 input or output
? means you can have 0 or more of the preceding item
+ means you can have 1 or more of the preceding item

Example:

```
operation <functionName>
{
        [input]? : <inputVariableName> <max min>+
        [output]? : <outputVaribleName> <max min>+
}+
```

Operation, input and output are keywords the parser uses to keep track of what it is parsing

### 6.3.3 The Perl BVA Generator Design

The BVA Perl script to extract test cases works as follows:

- Open the BVA specification file (BVASpecification.txt),
- Find "operation" keyword and the max and min values for defined by each input and output keyword in the BVA specification file. Write them to a new file with each input and output on a new line in the output file (bvaValues.txt),
- Open bvaValues.txt,
- For each line in bvaValues.txt,
    - create test inputs for the min value against all other min values of inputs and outputs
    - create test inputs for the max value against all other min values of inputs and outputs,
    - output the test inputs to "test_case_data.txt"
- open the "test_case_data.txt" file, search for duplicated lines and remove them, this removes duplicated tests cases.
- for each line in "test_case_data.txt"
    - get the number of clients to be created,
    - get the port to be used,
    - get the timeslot to be added,
    - get the timeslots to be used by this test
    - get the channels to be used
    - create the C code for the port configuration
    - create the C code structure for channels to be added to
    - for every client:

    - add a channel with the current timeslot to be used to the C code
    - check if all the test input satisfies the timeslots to be used on the current E1 line, increment the timeslot to be added if there are more timeslots required, otherwise increment the line. If the line is > 4 increment the port number and set the line to 0. Create a new port configuration C code structure; create a new channel configuration C code structure.
    - if we have added the max channels to the current port, increment the port number and set the line to 0. Create a new port configuration C code structure; create a new channel configuration C code structure.
    - if we are adding the last client, add all the remaining timeslots required to the last client incrementing the timeslot to be added each time, create new port configuration C code structures and new channel configuration C code structures when the boundaries are met.
    - Print a summary of the test case in the list of test see in Figure 7
- Once all test cases have test code generated close off the test case list structure
- Combine all the channel configurations and list of test cases into one file

The Perl script and the input and output of the above design are listed in Appendix V.

## *6.4 DU Coverage Test Case Generator*

### 6.4.1 Deciding how to generate the test code

Based on the experienced gained out of the BVA test code generation it was decided to persist with Perl in parsing the specification file to create test code. However due to the nature of DU testing it was not be possible to use the existing test framework.
Issues such as; the need for calls to the API out of its normal order and; repeated function calls does not fit naturally in the existing test API framework.

This design was based on the setup and teardown the software environments i.e. initialise and put back to start-up state after each test. If the test does not exit as expected we reset the system. Each test is independent code with no preconditions required based on execution of other code.

### 6.4.2 The DU Generator Design

This section describes the DU generator Perl script. The sections described here can be referenced as comments in the Perl file in Appendix VI.

The Perl script executes 4 main parts in sequential order. The 1st part is small in that it parses the specification file of the "global variables". Global variables are the DU pairs we wish to find in the specification.

In the second part of the script we look for each global variable in the operations provided in the specification file. We create an array of definitions and an array of uses and then we create all the DU combinations of these. Each combination is checked to see if it is a valid combination, we apply specific rules to judge validity; these rules are described in more detail below. From the valid combinations we can create skeleton code of where the function is first defined then used. As we can call the API in a specific order, we can fill in the skeleton code between the defined and use cases with the correct sequence of functions calls. We apply an algorithm that ensures that a variable is not re-defined before it is used. This ensures that we test the correct DU pair.

In the 3rd part we add variables at the top of the code which tracks the expected state of the system. We check these variables against pre-conditions of the functions to be called and set the expected results and set tracking variables based on the post condition of the functional call and the result expected of it.

In the final part of the script we replace pseudo code with compliable C code. We add an initialisation function for our tracking variables and we add test code that re-initialises the system state in case of unexpected results.

### 6.4.2.1 Part 1

Open the specification file and extract the global variables into a Perl array

### 6.4.2.2 Part2

This section of the Perl script is responsible for finding the DU cases (for the global variables found in the previous section) in the specification. It is also used to extract timeslots to be used in a test and ensuring that we keep track of which timeslots were added in a test. We also need to keep track which functions define or redefine a global variable so that when test code is created for only for valid DU pairs. This part of the script creates the duPsuedoCode.c file and the duCoverage.txt which lists all the valid and invalid DU pairs

For each global variable added to the Perl array in section 6.4.2.1:

- Open the specification file and find all instances of the global variable that is either defined ('ensures') or used ('requires'). The following rules have been applied in this part of the Perl script:
  - The keyword 'operation' is used as a marker, if any processing was being done on a 'requires' or 'ensures' we reset their flags. We clear the function parameters string so that we can add a new set of parameters for the new operation.
  - The keyword 'consumes' defines the parameter inputs to a function, for each consumes line we find within an operation we add it to the function parameters string. For example if we were parsing the operation in Figure 8, we read in 3 parameters, a file descriptor, the port number and the configuration being applied to the port.

```
operation portUp
{
        consumes: fileDesc
        consumes: portNum
        consumes: config
        alters: portConfig[portNum]
        produces: status
}
```

**Figure 8 - Example of operation parameters**

- If we find the keyword ‘requires’, we set the flag that we are processing a ‘requires’ line. If the flag is already set then we assume the ‘requires’ statement is already being processed. The flag is unset when we find a subsequent line that does not have the ‘\’ character as the end of the line. An example of this is shown in Figure 9. The 1st line of this figure we find a ‘requires’ keyword, we set the flag and the flag remains set until the 4th line were there is no ‘\’ at the end of the line. This is an example of a large conditional statement that applies to the timeslot map to check that it is valid. In this section of the Perl code we also apply a special rule that applies to lines 62-90 of our RESOLVE ‘like’ specification. On these lines we define valid maximum and minimum timeslot values for each line for single, dual and quad timeslot channels. The Perl code extracts the timeslot map which is used later when a channel is added

```
requires: (tsMap.line0_timeslot_bit_map >= 0x00000002 && tsMap.line0_timeslot_bit_map <=0xFFFFFFFF) || \
          (tsMap.line1_timeslot_bit_map >= 0x00000002 && tsMap.line1_timeslot_bit_map <=0xFFFFFFFF) || \
                    (tsMap.line2_timeslot_bit_map >= 0x00000002 && tsMap.line3_timeslot_bit_map <=0xFFFFFFFF) || \
                    (tsMap.line3_timeslot_bit_map >= 0x00000002 && tsMap.line0_timeslot_bit_map <=0xFFFFFFFF)
```

**Figure 9 - Multiple Line Requires Specification**

- If we find the keyword ‘ensures’, we set the flag that we are processing an ‘ensures’ line. If the flag is already set then we assume the ‘ensures’ statement is already being processed. The flag is unset when we find a subsequent line that does not have the ‘\’ character as the end of the line. We use this section to keep track of all functions which define the current global variable we are processing. This is used later when creating the test code so that we know not to call any functions that redefine our DU pair before it is used. We also need to make sure that if we are adding a channel that we keep track in our code of the timeslot that was added. This is useful if we want to test adding a channel again with the same timeslot, we can expect that the API would not allow such a case. When we find an ensures, we add to the pseudo code all the preceding functions that occur before the define operation. For example Figure 10 shows the pseudo code created when a global variable is defined (ensures) in PortUp and all the preceding functions open and init are called before it. Then ChannelAdd is the use (requires) function

```
Test Case 2(void)
{
init
open
portUp( fileDesc portNum config)

channelAdd( fileDesc portNum channelNum channelSize tsMap)
}
```

**Figure 10 - Psuedocode of defined in PortUp and Use in ChannelAdd**

- If we find a line with the global definition declared, it is treated as a definition. The pseudo code is added as init, which is the start up state of the system
- Lines 233-242 output the DU pairs found for the global variable
- Line 246 – 341 outputs the pseudo code for each DU pair, the pseudo code adds some test specific into such as timeslots to be used. If this is not added the default timeslot is used (timeslot 1 on line 1)

### 6.4.2.3 Part 3

This section re-opens the RESOLVE 'like' specification and extracts out all the pre and post condition test code for each operation in the specification. Preconditions are specified by any requires line of an operation and post conditions are specified by and defines line of an operation.

### 6.4.2.4 Part 4

The final section of the Perl script opens the pseudo code and from this creates the real test code. Init () in the pseudo code is replace with what the specification indicates is the initial state of global variables. It also sets some test input parameters to a default state, such as the 1st timeslot for a channel is always set to timeslot 1.

Some tests update this if a use case requires a different timeslot to be tested. Based on the pre and post condition code create in the previous section it is possible to determine if function calls will be accepted or not and if the result of the API call is as expected for each function call then the test passes. Code is added to the end of the test to bring the system back to its default state. This involved removing any open channels, bring down the port unitising the framer and closing the driver. If for any reason the resetting of the system functions fail then the state of the system is unknown and the tester is required to reset the system before running any more tests

## *6.5 Implementation Analysis*

This section analyses the efficiency of each technique in terms of the time taken to implement and test.

### 6.5.1 Requirements Driven Tests

The requirements driven tests were developed from a very early stage of the project. The usage model was not well documented and busy timelines by the developers meant that the system testers had to make their own estimations on the size and scope of the code required to implement the test framework. Also the HSS Voice test code framework was planned to be integrated with HSS Data test code framework so that test inputs work with either feature. This made the task difficult to implement due to limited availability of assistance from developers, and complicated to integrate a test framework to two features. The time taken to implement the framework was recorded for the combined HSS Voice and HSS Data test framework. This took 6 weeks, however the code divided roughly 50:50 to the two features so for comparative purposed the HSS Voice Driver tests took 3 weeks to implement. The author was responsible for the software architecture of the HSS Voice test framework. Implementation was done by a graduate student. The author later took on execution of tests on the system under tests. There were quite a few lessons learned when attempting to execute the combined feature test framework which resulted in a large churn of the test code framework. The result of this is that the author gained an in depth knowledge of the system and what is possible and what is not. The learning from executing and debugging the HSS Voice test code framework, made the task of implementing and executing the boundary Value analysis and DU test code generator much easier, so a time analysis of the each methods is somewhat skewed.

### 6.5.2 BVA Tests

The BVA test code was designed to run off the existing test framework so in effect 90% of the work for this was already done and we just had to write a script to extract test inputs from a specification and create the test data file. The time taken to implement the test code generator and run the tests was 20 hours, (~ 3 days work). However the test case generator only generated 29 test cases. The author feels that you could manually create the same code in 1 day. So the automated test generation of black box tests does not seem very time efficient. The test code generator heavily linked to the test framework. If this were to be applied to a different software product where the API is completely different then the test framework would have to be re-written and so would the test generator. You could use the general process of the BVA test generator script to create a new generator, but it does not seem worthwhile to try improving a task that only takes 1 day in the first place.

### 6.5.3 DU Test

The DU test code by its nature could not use the existing test framework. The generator was required to generate a completely new test application code. It took ~50 hours (~1.5 weeks) of work to create the Perl script and execute the test. The bulk of this work was in implementing the Perl script. The tests take about 1 hour to execute using an automated Perl script to call each test. The test execution used the automated framework that was already in place that ran the Requirements driven tests; it was simply a matter of editing the Perl script slightly to call the DU test code function calls instead of the Requirements driven test case function call and add some code to reboot the system if the test did exit gracefully.

Although it takes a long time to create the test generator script. The test code is complete and not dependent anything except the HSS Voice driver. This method was the most time efficient method of creating a test suite.

# 7 Test Results

The test cases for each method were run on an early build (build55) and then rerun on the first customer release of the package (version 1.0). We re-run the tests to see if one method is better the others at identifying defects early and to check if the fixing of defects from an early build does not affect the code in some other way in a later build. The early build is the first available package that is able to make all the kernel objects required for the HSS Voice Driver. It is possible to extract earlier versions than build55 from the version control system, but the T1E1 framer driver was not working to an extent that it was able to process traffic. The number of defects found for each method vs. each build is summarised in Table 2. Table 3 lists the type of issues found in each test run. The most common problem discovered by all techniques is that bytes could be missing or duplicated in a payload received back from the T1E1 Framer device. The number of bytes missing or duplicated appears to be directly related to the size of the channel, 1 byte repeated or missing for 80 byte channels, 2 bytes repeated or missing for 160 byte channels and 4 bytes missing of repeated for 320 byte channels. This observation has been categorized into 1 defect for counting purposes in Table 2.

| | Tests Run | Defects Found | |
|---|---|---|---|
| | | Build55 | Version 1.0 |
| Requirements driven test Method | 13 | 3 | 0 |
| boundary Value Analysis | 29 | 2 | 0 |
| DU tests | 89 | 2 | 1 |

**Table 2 – Defect Count for each test method**

Table 3 below shows the defects found in each method and on each version of the software it was run against ….

| | Defects Found | |
|---|---|---|
| | Build55 | Version 1.0 |
| Requirements driven test Method | Found Issue of 2 bytes **repeated** in a Framer Loopback test | No Defects Found |
| | Found Issue of 4 bytes **repeated** in a Framer Loopback test | |
| | Found Issue of 4 bytes **missing** in Framer Loopback test | |
| | External Traffic does not work | |
| | Analog and Framer does not work together on a system | |
| boundary Value Analysis | Found Issue of 1 bytes **repeated** in a Framer Loopback test | No Defects Found |
| | Found Issue of 2 bytes **repeated** in a Framer Loopback test | |
| | Found Issue of 4 bytes **repeated** in a Framer Loopback test | |
| | Found Issue of 1 bytes **missing** in a Framer Loopback test | |
| | Found Issue of 2 bytes **missing** in a Framer Loopback test | |
| | Found Issue of 4 bytes **missing** in a Framer | |

| | | |
|---|---|---|
| | Loopback test<br>Firmware Error Reported bringing up 128 channels – system reboot required<br>Stability Issue when removing 128 channels[5] | |
| DU tests | When there are channels still associated to the device, Close returns success when it should return fail | When there are channels still associated to the device, Close returns success when it should return fail |
| | Found Issue of 1 bytes **repeated** in a Framer Loopback test | |
| | Found Issue of 1 bytes **missing** in a Framer Loopback test | |

**Table 3 - Summary of Defects Found**

[5] This issue was only observed once and could not be repeated in subsequent tests

# 8 Analysis of Results

The follow sub sections analyse each of defects discovered by the test techniques employed as list in Table 3 - Summary of Defects Found.

## *8.1 Combined Usage Of Analog and Framer Mezzanines*

The results listed in section 7 are relatively limited and at first glance do not show any clear choice of a preferred test method. The requirements driven test cases reveal 1 more defect than the BVA or DU tests. This extra defect was found by a test that we explicitly stated in BVA or DU that we would not test nor compare against the Requirements driven test cases i.e. tests that use the Analog mezzanine. However it does show the importance of combinations of features working together and it is felt by the author that BVA and DU methods may not have found this issue of a T1E1 framer and the Analog Mezzanine working together at the same time. Both the BVA and DU methods are not capable of generating a test case with such a combination, they either test with one or the other. It is possible that a tester could run all Analog mezzanine tests on one day, then the next day come in change the physical setup on the system by swapping the Analog mezzanine with a T1E1 framer and run framer tests on the new nightly build. If on the same night the developer makes a change to the driver that results in the Framer working and the Analog card not working then the defect would go unnoticed for a while, unless a test exists that use both cards in one test. In reality this is exactly what happened, the development team whom each were responsible for testing their own driver, did not have tests that use a combination of mezzanines cards. Whilst the very definition of system tests means all features should be tested working together. This seems to be a major drawback of the BVA and DU methods that it is not capable to create such a scenario. We would have to extend BVA to combinational BVA and extern DU to all DU paths to find such combinations.

## *8.2 Missing/Repeated Bytes*

The results tables in section 7 tables does not show is that the boundary value analysis test were better able to find the distribution of bytes missing/bytes repeated in the tests, it also makes it clear that the number of bytes missing or received is tied to the channel size, this can been seen in the detailed results in Appendix IX. The BVA result from build 55 also leaves an open issue: why do single timeslot channels sometimes fail with repeated bytes and sometimes fail with missing bytes. We decided to run 2 tests repeated 20 times over the resulted are tabulated in Table 4

| BVA testId | Number of Times Bytes were missing | Number of Times Bytes were repeated | Number of times test passes |
|---|---|---|---|
| 22 | 7 | 4 | 9 |
| 26 | 8 | 5 | 7 |
| Probability of Occurrence | 37.5% | 22.5% | 40% |

**Table 4 - Incidence of Repeated and Missing Bytes in Build55**

The pattern is not obvious in the results from Requirement driven tests or the DU tests. This could be useful information to help development root cause the issue.

## 8.3 Straight Through Traffic on Framer not working

The requirements driven test cases found that the Framer did not work in normal throughput mode. The BVA or DU methods could also have found this defect if repeated using Framer throughput.

## 8.4 Bring up of 128 channels Causing Firmware Error

There was a problem encountered in Framer loopback tests when bringing up a test that required 128 channels. Whilst the Requirements driven tests and BVA tests had such test cases the problem was not identified in the requirements driven test case. Requirements driven test for this case used external traffic with CPU loopback whilst the BVA test case used internally generated traffic with framer loopback. It should also be noted that even though external traffic was not possible in build55. It was still possible to bring up all 128 channels. The reason that BVA found this issue was traced down to differences in the test code framework. Both methods use the same test code however difference functions are uses for looping back external traffic and generating/checking internal traffic. The function that processed external traffic when bringing up the channels has a small delay in the code after each channel was brought up. The code for generating traffic had no such delay. This defect cannot be attributed to BVA analysis alone as it was not a particular boundary the resulted in the defect rather is was cause by calling the channelUp function repeatedly to fast. If the test framework had been consistent both methods could have found such a defect.

## 8.5 Stuck in loop closing channels

One of the BVA tests discovered an issue when removing channels. It appeared to get stuck on continually sending messages to the Programmable IO unit to remove the channel. This issue was not reproducible on subsequent re-runs of the test. It was not considered in the count of defects in Table 3. Such a defect would likely be exposed on a stress or reliability test.

## 8.6 Close not returning Error when channels are still open on the device

The DU test cases found one issue that was unique to this method. It also highlights the importance of a documented API. In this instance the error was due to Linux and it is not clear from the documentation why Linux in this case returns the value it returns. Some test cases looked at DU combinations that required portDown to be called before channelRemove. In such case the portDown call was expected to fail because the API specifies that all channels on all clients need to be removed from a port before it can be closed. If a function call returns the expected result then the function that normally follows is called, which in the above case is close. Close is also expected to fail because the channel still exists on the client. However Close returned success when it was expected not to. Further inspection of this problem revealed that the HSS Voice driver returns an error code to Linux which then returns success to the user but closes the file handle! With a closed file handle the channels are tied up in the system and cannot be accessed without rebooting the system. This seems to be a problem with Linux itself rather than the HSS Voice driver

## 8.7 Summary of Analysis

Of the defects found, all could have been found by currently employed test selection method. It was only the implementation of those tests that resulted in them not being all found by the current method. The defects found by DU and BVA methods are defects that those methods were not specifically intended to find. For example no defects were found at boundaries by the BVA method and no defects were found as a result the state of the system the DU tests placed it in. The defects found were either the cause by:

1. Timing errors, possibly internal interrupt related threads causing lock up
2. A collection of drivers working together, such as the analog and framer driver in one system or,
3. Internal processing that missed or duplicate chunks of data

It appears from this analysis that the current method is capable of finding items 2 & 3. As a system test, it is essential to check that all components work together. The current method also found the duplicated and missing data of item 3, by verifying the payload sent was the payload received back, any test of any method should check this. The current requirements driven method is not so well targeted at timing errors. The hope is, that by running test for extended periods ~24 to 48 hours that timing errors would expose themselves. But such tests are time consuming and timing related errors as found in the test we ran can be found within minutes. This suggests that a test selection technique that targets timing errors could be beneficial in finding errors in such a system.

# 9 Conclusion

We investigated the current methods of test selection and automation based on concerns being raised from previous projects; is this test strategy good enough? Can we do better? We cannot measure our levels of success unless we have something to compare it to. Based upon this we investigated other black box testing techniques. We looked at how other people measure the effectiveness of there testing and we looked at test methods that lend themselves well to test case generation with the idea that automatic generation of test cases would decrease test development times.

We selected the boundary value analysis method for it ease of implementation and for its ability of find a well known source of faults. We derived a method of black box test case generation based on white box DU testing. We look the principles that Edwards [15] describes in using RESOLVE to generate DU test cases. We did this by developing a new method based on specifying the API in a certain form then being able to parse this specification using a Perl script to create C test code that is easily automatable.

Finally we compared the above two methods of generating tests against the existing set of test cases by running them on an early version of the software and a later version of the software to see if any method was better at finding defects. We also discussed the implementation time of each method to access the cost of development for each methods.

In the following subsections we provide concluding comments on each method then discuss the lessons learnt from this investigation and what future work we could do going forward

## *9.1 Requirements Driven Method*

The requirements driven test cases provide a limited detail in testing but it covers a broad range of combinations which is essential when testing a system. The method requires experiences engineers in order to be effective. The defects found in this method seem to reflect system issues rather than defects that could be found by integration or unit testing alone. As mentioned in our introduction, this method is seen as costly and time consuming method of test development, due to all the manual work of test coding and automation of equipment.

## 9.2 BVA Method

The BVA test cases caused most of the tests to be focused on timeslot allocation. So whilst the BVA generated a reasonable amount of test cases and found defects, it did not find the defects that the method was designed to find. The main reason for this is that the development team are responsible for boundary checks so if they have done a good job at creating their test cases then we should not find any such defects; this turned out to be true. The effort at automatically creating test cases was very fast, however it piggybacked the existing test code infrastructure. The test code infrastructure is setup to allow easy add and removal of tests. It one were to separate out the time it took to add the requirements driven tests to the test infrastructure then the BVA method was not any more time effective and it did not find any boundary related defects. The method is easy to implement and could be useful if the project to be tested had engineers of limited testing experienced assigned to it.

## 9.3 DU Method

The DU method generated the most test cases, most of these focused on checking that that the sequence of calls returned the correct result. This method did not find any problems with the HSS Voice driver on return values or out of order calls. Similar to BVA, out of order calls and return checking is in the domain of integration and unit test code and this is likely the main reason why this test method did not find the defects that it was targeted to find. The method looks very interesting in terms of time efficient method of generating tests. Each test is self contained not dependent on external test equipment and easily automatable. This method seems highly suitable to be used at the integration test level. The largest amount of effort is ensuring that the Perl script parses the specification correctly to create the correct code. It is also very valuable if the API were to change, which has happened in other projects. Instead of changing thousands of lines of code, all one has to do is to change the specification file and adjust the Perl scrip to re-generate the tests.

## 9.4 Automatic Test Case Generation

The RESOLVE like specification was used in conjunction with a script implemented in PERL which shows that it is possible to automatically create test cases. This method showed to be more time effective in generating test cases that the requirements driven method and BVA method. Also there is less chance of errors in the code that is generated. The script is to generate the test cases is complex and requires and experienced software engineer to implement. Whilst the test cases generated did not find significant numbers of defects. It could be quite useful means of generating test cases for the development team, as it can test that the API is handling the input of data correctly, returning the correct values and that the system is in the correct state according to the specification. The test cases generated were also found easy to plug into and automated execution system, mainly due to the fact that each test case contains code that setups the system then returns it to its start-up state, so that the next test case to be executed is not dependent on the results of the previous.

## 9.5 Discussion on the Relative Effectiveness

We looked at several papers on test effectiveness. Frankl [2] indicated the more lines of code coverage the more chance of finding defects. By simply implementing all the test we have across the three methods we achieved more code coverage (although we did not measure it) and we have found more defects.

In Huber [6], he suggested the use of metrics as a measure of effectiveness. We re-used our existing test framework for BVA tests, and we applied ~ 3 days effort to add BVA tests to find 1 additional defect. This is 20% extra effort[6] resulted in 25% more defects[7]. The DU testing effort took 7.5 days for 1 additional defect. The equates to 50% extra effort for 25% more defects and the defect found in this case was not due to the Intel software. So based on this one has to question the value that the DU testing added. It seems a very good method for automatic generation of test cases. However it is be better suited to integration testing.

[6] Original requirements driven estimate of 15 days was discussed in section 6.5.1

[7] BVA found 1 new defect unique to its method refer to section 7.

Bertoloni [10] suggested that a combination of methods is better to find more defects as each method targets different sources of defects. From our results we cannot confirm whether it was the combination of methods that result it more defect being found or whether it was due to simply more tests = more defects as suggested by Frankl [2].

If you look at the results it appears that the number of tests run is directly related to the number of defects found and based on this we should develop more tests. We could use the BVA method combined with the requirements driven method, this would give us more tests for less effort. However extra defect found by BVA was more due to inconsistent implementation in our test framework. The above mentioned effort vs. defect found metric is somewhat skewed. There is simply not enough in the results to make the assumption that BVA added value. If any value has been added it is marginal at best.

## 9.6 Future work

This investigation has exposed the benefits of analysing the cause of all defects and it has highlighted causes that our methods are not specifically targeting. Some of the defects we identified appeared to due to timing related issues. It would be prudent to investigate a method of test selection that targets timing related issues. Also the system test team should be measuring its relative success from project to project. Effort spend vs. defects found, if this number were to be dropping then the test team should investigate why this is so; are developers writing better code, maybe they have done there own defect analysis and they have improved coding areas were system test traditionally found bugs. The system test team would have to changes its test selection method if it was no longer being successful at find defects.

We also discovered the issue of how to treat bitmasks in Boundary Value Analysis was undocumented. This is clearly an area which could be investigated further, as the BVA method is very popular for all types of software projects. The use of bitmasks is very common in embedded software.

## *9.7 Final Thoughts*

In terms of defects found by each method the requirements driven (adhoc) test selection method is as effective as more rigorous methods of test selection. A more rigorous method could be easily applied by an engineer with limited experience and more tests could be added based on experiences testers input. Testing of embedded software is complicated by the fact that a lot of activity happens within the system that is not immediately visible to the user, hence a large amount of effort could be spent testing for few defect founds. It appears to be a situation the more tests you run the more defects you will find. The number of defects found is too small to define statistical differences between the methods. The requirements driven tests could likely have found all the defects that the other methods found (with the exception of the close with active channels), if the test framework implementation was more consistent. However the BVA and DU tests by nature would not have found all the defects that the requirements driven tests found. The combined use of drivers seem to be the main cause of defects. Also, the BVA and DU method appear to be designed to find defects that are in the domain of responsibility of the development team. In retrospect assuming that the development team have a good test selection criterion to cover return checking, boundary values and out of order calls, then there is no reason to believe that BVA or DU test selection methods would be good at finding defects once the software is available to the system test team.

In terms of cost benefit, the DU method was more time efficient at generating test cases. However as we have mentioned, the test cases generated maybe more useful as integration level tests rather than system tests. The parser to generate the test cases is quite complicated to apply so the DU method does require an experienced software engineer. Many hours was spent implementing the Perl script that generates the test cases, however once done, we have created a reasonable set of test cases that are independent of each other, they each return the system to its start up state and they are easy to automate. The DU method does have its appeals and is a useful method not only for Integration level tests Intel Shannon, but to the wider test community interested in generating test cases this way.

The BVA method was similar to the current requirements driven method in terms of timeliness, however it is methodical and easy to apply so it could be useful if there are shortages of engineers on a project and somebody of limited testing experience is assigned the job of developing tests, then they could apply this method.

This gives us some confidence that the current method of test selection is acceptable, but the potential for room for improvement has been flagged but possibilities in test case generation and other test selection methods to try.

# 10 References

[1]. Panagiotis Louridas, "JUnit: Unit testing and Coding in Tandem," *IEEE Software*, vol. 22, no. 4, pp. 12-15, Jul/Aug, 2005

[2]. Phyllis G. Frankl, "Assessing and enhancing software testing effectiveness" ACM SIGSOFT Software Engineering Notes, Vol 25 , Issue 1, Jan 2000, Pages: 50 - 51

[3]. Parikh, Keyur; Kim, Junius, "TDM Services over IP Networks", IEEE Explore**,** 29-31 Oct. 2007 Page(s):1 – 10

[4]. Tsong Yueh Chen, Fei-Cheng Kuo, Robert Merkel, "On the statistical properties of testing effectiveness measures", Journal of systems & Software, Vol 79, Issue 5, 2006, pages 591 – 601

[5]. T.Y. Chen, H. Leung & I.K. Mak, "Adaptive Random testing", Lecture Notes in Computer Science, Volume 3321/2005, pages 320-329

[6]. Jon T. Huber, "Efficiency and Effectiveness Measures To Help Guide the Business of Software testing", SM/ASM Conference, 1999

[7]. Kwang Ik Seo, Eun Man Choi, "Comparison of Five Black-box testing Methods for Object-Oriented Software", Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications. Pages: 213 – 220, 2006

[8]. Beizer, Boris, "Software testing Techniques", 2nd Edition, Van Nostrand Reinhold, 1990.

[9]. Beizer, Boris, "Black Box testing", John Wiley and Sons, 1995

[10]. Bertolino, Antonia "Software testing Research: Achievements, Challenges, Dreams", Future of Software Engineering, 2007. FOSE '07, 23-25 May 2007 Page(s):85 – 103

[11]. J. Wegener and M. Grochtmann. "Verifying timing constraints of real-time systems by means of evolutionary testing." Real-Time Syst., 15(3):275–298, 1998.

[12]. Dr. Velur Rajappa, Arun Biradar, Satanik Panda, "Efficient Software test Case Generation Using Genetic Algorithm Based Graph Theory", First International Conference on Emerging Trends in Engineering and Technology, 2008 , pages 298-303

[13]. A. Z. Javed, P. A. Strooper, G. N. Watson, "Automated Generation of test Cases Using Model-Driven Architecture," *Automation of Software* test*, Second International Workshop on*, vol. 0, no. 0, pp. 3, Second International Workshop on Automation of Software test (AST '07), 2007.

[14]. B.-Y.Tsai, S. Stobart and N.Parrington, "Employing data flow testing on object-oriented classes", IEE Proceedings - Software -- April 2001 -- Volume 148, Issue 2, p. 56-64

[15]. Stephen H. Edwards, "Black-box testing using flowgraphs: an experimental assessment of effectiveness and automation potential", Software testing, Verification and Reliability, vol 10, issue 4, pages: 249-262, YR: 2000

[16]. Wood, M., Roper, M., Brooks, A., and Miller, J. 1997. Comparing and combining software defect detection techniques: a replicated empirical study. In Proceedings of the 6th European Conference Held Jointly with the 5th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Zurich, Switzerland, September 22 - 25, 1997)

[17]. Marc Roper. Software testing. McGraw-Hill, 1993

[18]. Sitaraman M, Weide BW, . Component-based software engineering using RESOLVE. ACM SIGSOFT Software Enigineering Notes, 1994, 19(4): 21-67.

[19]. Luay Ho Tahat, Atef Bader, Boris Vaysburg, Bogdan Korel. "Requirement-Based Automated Black-Box test Generation", Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, Pages: 489 – 495, 2001

[20]. H. V. Kantamneni, S. R. Pillai, and Y. K. Malaiya. "Structurally Guided Black Box testing". Technical report, Dept. of Computer Science, Colorado State University, Ft. Collins, CO, USA, 1998

[21]. Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology Linux* Device Driver API Reference Manual, September 2008, Reference Number: 320416, Revision -001, http://download.intel.com/design/intarch/ep80579/320416.pdf

[22]. The Eclipse Foundation, "Business Intelligence and Reporting Tools", 2009, http://www.eclipse.org/birt/phoenix/

[23]. IBM, "Rational ClearQuest Test Manager", http://www-01.ibm.com/software/awdtools/clearquest/index.html

# Appendices

## I. HSS Voice API

The HSS Voice API is described in section 5 of the following link:

Intel® EP80579 Software for IP Telephony Applications on Intel® QuickAssist Technology Linux* Device Driver API Reference Manual, September 2008, Reference Number: 320416, Revision -001

http://download.intel.com/design/intarch/ep80579/320416.pdf

## II. Boundary Value Analysis Input/Output Range Specification file

The following text specifies the operations, inputs and outputs and their maximum and minimum values used and extracted to create test cases for the boundary Value Analysis method:

```
operation open
{
        output: clients 1 128
}
operation portUp
{
        input: portNum 0 2
}
operation channelAdd
{
        input: portNum 0 2
        input: channelId 0 127
        input:          tsMap          0x00000002,0x00000000,0x00000000,0x00000000
0x80000000,0x00000000,0x00000000,0x00000000
0x00000000,0x00000002,0x00000000,0x00000000
0x00000000,0x80000000,0x00000000,0x00000000
0x00000000,0x00000000,0x00000002,0x00000000
0x00000000,0x00000000,0x80000000,0x00000000
0x00000000,0x00000000,0x00000000,0x00000002
0x00000000,0x00000000,0x00000000,0x80000000
0x0000000C,0x00000000,0x00000000,0x00000000
0xC0000000,0x00000000,0x00000000,0x00000000
0x00000000,0x0000000C,0x00000000,0x00000000
0x00000000,0xC0000000,0x00000000,0x00000000
0x00000000,0x00000000,0x0000000C,0x00000000
0x00000000,0x00000000,0xC0000000,0x00000000
0x00000000,0x00000000,0x00000000,0x0000000C
0x00000000,0x00000000,0x00000000,0xC0000000
0x0000001E,0x00000000,0x00000000,0x00000000
0xF0000000,0x00000000,0x00000000,0x00000000
0x00000000,0x0000001E,0x00000000,0x00000000
0x00000000,0xF0000000,0x00000000,0x00000000
0x00000000,0x00000000,0x0000001E,0x00000000
0x00000000,0x00000000,0xF0000000,0x00000000
0x00000000,0x00000000,0x00000000,0x0000001E
0x00000000,0x00000000,0x00000000,0xF0000000
        output:         tsUsed         0x00000000,0x00000000,0x00000000,0x00000000
0xFFFFFFFE,0xFFFFFFFE,0xFFFFFFFE,0xFFFFFFFE
        output: channelsUsed 1 128
}
end;
```

# III. RESOLVE 'Like' Specification

```
class HSSVoice
        portStatus: int [3]
        channelsOnPort: int [3]
        channelsOnFd: int [128]
inserted: bool
        openCounter: int
        channelsUsed: int
        tsUsed: icp_hssdrv_timeslot_map_t
        portConfig: int [3]
        channelStatus: char [128]
        channelConfig: icp_hssdrv_timeslot_map_t [128]
        readBuffer: char [40960]
        writeBuffer: char [40960]
        tsMap: icp_hssdrv_timeslot_map_t
;

init
{
        alters: inserted
}
#requires: inserted= false    #used
#ensures: inserted=true               #defined

operation open
{
        alters: openCounter
        produces: fileDesc
}
#preconditions
requires: openCounter<128
#postconditions
ensures: openCounter+=1

operation portUp
{
        consumes: fileDesc
        consumes: portNum
        consumes: config
        alters: portConfig[portNum]
        produces: status
}
#preconditions
requires: portConfig[portNum]==config || \
        portConfig[portNum]==NOT_SET
#postconditions
ensures: portConfig[portNum]=config
ensures: portStatus[portNum]=UP
```

```
operation channelAdd
{
        consumes: fileDesc
        consumes: portNum
        consumes: channelNum
        consumes: channelSize
        consumes: tsMap [4][32]
        alters: tsUsed
        alters: channelsUsed
        produces: status
}
requires: channelsOnFd[openCounter]<128
requires: portStatus[portNum]==UP
#requires: portConfig[portNum]=QMVIP iff tsMap >,0xFFFFFFFF
requires: (channelsUsed & channelNum) == 0 /*channel number is not used*/
requires:  (((tsUsed.line0_timeslot_bit_map  &  tsMap.line0_timeslot_bit_map)  ==0)  &&
((tsUsed.line1_timeslot_bit_map     &     tsMap.line1_timeslot_bit_map)     ==0)     &&
((tsUsed.line2_timeslot_bit_map     &     tsMap.line2_timeslot_bit_map)     ==0)     &&
((tsUsed.line3_timeslot_bit_map & tsMap.line3_timeslot_bit_map) ==0))
requires: (tsMap.line0_timeslot_bit_map >= 0x00000002 && tsMap.line0_timeslot_bit_map
<=0xFFFFFFFF) || \ /*ensure that the channel does not span E1's*/
               (tsMap.line1_timeslot_bit_map         >=         0x00000002          &&
tsMap.line1_timeslot_bit_map <=0xFFFFFFFF) || \
               (tsMap.line2_timeslot_bit_map         >=         0x00000002          &&
tsMap.line3_timeslot_bit_map <=0xFFFFFFFF) || \
               (tsMap.line3_timeslot_bit_map         >=         0x00000002          &&
tsMap.line0_timeslot_bit_map <=0xFFFFFFFF)
requires:   (tsMap.line0_timeslot_bit_map==0   &&   tsMap.line1_timeslot_bit_map==0   &&
tsMap.line2_timeslot_bit_map==0  &&  tsMap.line3_timeslot_bit_map==0x00000002)  ||  \  /*
can be 1Timeslot channels */
               (tsMap.line0_timeslot_bit_map==0  &&  tsMap.line1_timeslot_bit_map==0  &&
tsMap.line2_timeslot_bit_map==0 && tsMap.line3_timeslot_bit_map==0x80000000) || \
               (tsMap.line0_timeslot_bit_map==0  &&  tsMap.line1_timeslot_bit_map==0  &&
tsMap.line2_timeslot_bit_map==0x00000002 && tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0  &&  tsMap.line1_timeslot_bit_map==0  &&
tsMap.line2_timeslot_bit_map==0x80000000 && tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0                                       &&
tsMap.line1_timeslot_bit_map==0x00000002    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0                                       &&
tsMap.line1_timeslot_bit_map==0x80000000    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0x00000002                              &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0x80000000                              &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0) || \
               (tsMap.line0_timeslot_bit_map==0  &&  tsMap.line1_timeslot_bit_map==0  &&
tsMap.line2_timeslot_bit_map==0   &&   tsMap.line3_timeslot_bit_map==0x00000006)   ||   \
/*can  2Timeslot channels where timeslots are adjacent*/
```

```
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0 && tsMap.line3_timeslot_bit_map==0xC0000000) || \
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0x00000006 && tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0xC0000000 && tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0                                   &&
tsMap.line1_timeslot_bit_map==0x00000006    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0                                   &&
tsMap.line1_timeslot_bit_map==0xC0000000    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0x00000006                          &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0xC0000000                          &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0   &&   tsMap.line3_timeslot_bit_map==0x00060006)   ||   \
        /*can  be  4  timeslot  channels  with  16  bit  seperation  between  two  Timeslot
pairs*/
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0 && tsMap.line3_timeslot_bit_map==0xC000C000) || \
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0x00060006 && tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0 && tsMap.line1_timeslot_bit_map==0 &&
tsMap.line2_timeslot_bit_map==0xC000C000 && tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0                                   &&
tsMap.line1_timeslot_bit_map==0x00060006    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0                                   &&
tsMap.line1_timeslot_bit_map==0xC000C000    &&    tsMap.line2_timeslot_bit_map==0    &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0x00060006                          &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0) || \
                (tsMap.line0_timeslot_bit_map==0xC000C000                          &&
tsMap.line1_timeslot_bit_map==0       &&       tsMap.line2_timeslot_bit_map==0       &&
tsMap.line3_timeslot_bit_map==0)
#postconditions
ensures: channelsOnFd[openCounter]++
ensures:                         channelConfig[channelNum].line0_timeslot_bit_map=
tsMap.line0_timeslot_bit_map
ensures:                         channelConfig[channelNum].line1_timeslot_bit_map=
tsMap.line1_timeslot_bit_map
ensures:                         channelConfig[channelNum].line2_timeslot_bit_map=
tsMap.line2_timeslot_bit_map
ensures:                         channelConfig[channelNum].line3_timeslot_bit_map=
tsMap.line3_timeslot_bit_map
ensures: tsUsed.line0_timeslot_bit_map |= tsMap.line0_timeslot_bit_map
```

```
ensures: tsUsed.line1_timeslot_bit_map |= tsMap.line1_timeslot_bit_map
ensures: tsUsed.line2_timeslot_bit_map |= tsMap.line2_timeslot_bit_map
ensures: tsUsed.line3_timeslot_bit_map |= tsMap.line3_timeslot_bit_map
ensures: channelsUsed |= channelNum
ensures: channelsOnPort[portNum]++

operation channelUp
{
        consumes: fileDesc
        consumes: channelNum
        alters: channelStatus[channelNum]
}
requires: (channelsUsed & channelNum) >0 /*ensure that channel was added first*/
requires: channelStatus[channelNum]==DOWN
#postconditions
ensures: channelStatus[channelNum] = UP

operation read
{
        consumes: fileDesc
        alters: readBuffer
        produces: bytesRead
}
#postconditions
ensures:  sizeof(readBuffer)  ==  sumof  (for  i=0;  i<channelsOnFd[openCounter];  i++)
{channelConfig[fd[openCounter]][i].channelsize + 4}
ensures: readBuffer=writeBuffer iff portConfig[portNum] == NPE_LOOPBACK

operation write
{
        consumes: fileDesc
        alters: writeBuffer
        produces: bytesWritten
}
ensures:  sizeof(bytesWritten)  ==  sumof  (for  i=0;  i<channelsOnFd[openCounter];  i++)
{channelConfig[fd[openCounter]][i].channelsize + 4}

operation channelDown
{
        consumes: fileDesc
        consumes: channelNum
        alters: channelStatus[channelNum]
}
requires: (channelsUsed & channelNum) >=0 /*ensure that channel was added first*/
requires: channelStatus[channelNum] == UP
#postconditions
ensures: channelStatus[channelNum] = DOWN


operation channelRemove
{
        consumes: fileDesc
        consumes: channelNum
        alters: tsUsed
        alters: channelsUsed
        alters: tsUsedInChannel[channelNum]
        produces: status
}
requires: channelStatus[channelNum]==DOWN
requires: (channelsUsed & channelNum)  >0
#postconditions
ensures: channelsOnFd[openCounter]--
ensures: channelsOnPort[portNum]--
ensures: channelsUsed ^= channelNum
ensures:                        tsUsed.line0_timeslot_bit_map                          ^=
channelConfig[channelNum].line0_timeslot_bit_map
ensures:                        tsUsed.line1_timeslot_bit_map                          ^=
channelConfig[channelNum].line1_timeslot_bit_map
ensures:                        tsUsed.line2_timeslot_bit_map                          ^=
channelConfig[channelNum].line2_timeslot_bit_map
ensures:                        tsUsed.line3_timeslot_bit_map                          ^=
channelConfig[channelNum].line3_timeslot_bit_map

operation portDown
{
        consumes: portNum
        alters: portStatus
        produces: status
}
requires: channelsOnPort[portNum]==0
#postconditions
ensures: portStatus[portNum]=0

operation close
{
        consumes: fileDesc
        alters: openCounter
        produces: status

}
requires: openCounter>0
requires: channelsOnFd[openCounter]==0
#postconditions
ensures: openCounter--
```

## IV. Requirements Driven Test Cases

| test Identifier | Description | test to be Run | Changes required |
|---|---|---|---|
| systest_HssVoiceDrv_1_01 | NPE Looback: Verify Timeslot Channel Mapping Internal Loopback | no | Can be changed for Framer loopback but is the same configuration as systest_HssVoiceDrv_1_11 |
| systest_HssVoiceDrv_1_02 | NPE loopback: Verify correct behaviour of 2TS channel | yes | Changed to Framer Loopback |
| systest_HssVoiceDrv_1_03 | NPE loopback: Verify correct behaviour of 4TS channel | yes | Changed to Framer Loopback |
| systest_HssVoiceDrv_1_04 | Framer Loopback Verify correct behaviour of 4TS channels (1 T1) | no | Can be changed to E1, but would be the same a systest_HssVoiceDrv_1_04 |
| systest_HssVoiceDrv_1_05 | Framer Loopback: Verify correct behaviour of 2TS channels (1 T1) | no | Can be changed to E1, but would be the same a systest_HssVoiceDrv_1_03 |
| systest_HssVoiceDrv_1_06 | Framer loopback Verify correct behavior of 1 TS channel (1 T1) | no | Can be changed for E1 but is the same configuratio as systest_HssVoiceDrv_1_11 |
| systest_HssVoiceDrv_1_07 | NPE Loopback: TX idle pattern test | no | Tx idle is not being compared between test methods s this test will not be run |
| systest_HssVoiceDrv_1_08 | IA Loopback: Data Inversion Check | no | Data Inversion is not being compared between tes methods so this test will not be run |
| systest_HssVoiceDrv_1_09 | IA Loopback: Byte Swap test | no | Byte Swap is not being compared between tes methods so this test will not be run |
| systest_HssVoiceDrv_1_10 | IA Loopback Blocking: TX overflow test: (1Analog) | no | Tx overflow is not being compared between tes methods so this test will not be run |
| systest_HssVoiceDrv_1_11 | IA Loopback: Verify Timeslot Channel Mapping (1 E1) | yes | ok |
| systest_HssVoiceDrv_1_12 | IA Loopback Blocking: Verify correct behaviour of 2TS and 4TS channels (1 Analog) | yes | |
| systest_HssVoiceDrv_1_13 | IA loopback Voice and HDLC channel test (1 E1) | no | mix of voice and data channels not being compared t other tests methods so this test will not be run |
| systest_HssVoiceDrv_1_14 | IA loopback Blocking: Out of Order narrowband channel test (1E1) | yes | |
| systest_HssVoiceDrv_1_15 | IA loopback Blocking: Out of Order 16 bit linear channel test (1 Analog) | yes | |
| systest_HssVoiceDrv_1_16 | IA loopback Blocking: Out of Order & non-uniform channel test (2 Analog) | yes | |
| systest_HssVoiceDrv_1_17 | IA loopback Blocking: Out of Order & non-uniform channel test (1 Analog) | yes | |
| systest_HssVoiceDrv_1_18 | IA loopback Blocking: Maximum Wideband channels (3 Analog) | yes | |
| systest_HssVoiceDrv_1_19 | IA Loopback Blocking HSS Bypass test (1 Analog) | no | Hss Bypass not being tested by other tests Methods |
| systest_HssVoiceDrv_1_20 | IA Loopback Blocking 24 Narrowband channels on (1 T1) | no | T1 not being tested by other test methods |
| systest_HssVoiceDrv_1_21 | IA to Framer Loopback Blocking - Wideband Channels on E1 | yes | |

| | | | |
|---|---|---|---|
| systest_HssVoiceDrv_1_22 | IA to Framer Loopback Blocking - Wideband Channels on T1 | no | |
| systest_HssVoiceDrv_1_23 | IA Loopback Blocking - Mixed HSS Mode - 4 Analog, 96 T1, 28 E1 channels (1 Analog, 2 E1's) | no | Mixed mezzanine configurations not being tested b other tests |
| systest_HssVoiceDrv_1_50 | IA Loopback NonBlocking: TX overflow test: (1Analog) | no | Tx overflow is not being compated between tes methods so this test will not be run |
| systest_HssVoiceDrv_1_52 | IA Loopback NonBlocking: Verify correct behaviour of 2TS and 4TS channels (1 Analog) | no | this is a repeat of systest_HssVoiceDrv_1_12 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_54 | IA loopback NonBlocking: Out of Order narrowband channel test (1E1) | no | this is a repeat of systest_HssVoiceDrv_1_14 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_55 | IA loopback NonBlocking: Out of Order 16 bit linear channel test (1 Analog) | no | this is a repeat of systest_HssVoiceDrv_1_15 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_56 | IA loopback NonBlocking: Out of Order & non-uniform channel test (2 Analog) | no | this is a repeat of systest_HssVoiceDrv_1_16 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_57 | IA loopback NonBlocking: Out of Order & non-uniform channel test (1 Analog) | no | this is a repeat of systest_HssVoiceDrv_1_17 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_58 | IA loopback NonBlocking: Maximum Wideband channels (3 Analog) | no | this is a repeat of systest_HssVoiceDrv_1_18 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_1_59 | IA Loopback Non Blocking HSS Bypass test (1 Analog) | no | Hss Bypass not being tested by other tests Methods |
| systest_HssVoiceDrv_1_60 | IA Loopback Non-Blocking 24 Narrowband channels on T1 | no | T1 not being tested by other test methods |
| systest_HssVoiceDrv_1_61 | IA to Framer Loopback Non-Blocking - Wideband Channels on T1 | no | T1 not being tested by other test methods |
| systest_HssVoiceDrv_1_62 | IA to Framer Loopback Non-Blocking - Wideband Channels on E1 | no | same as 1_22 |
| systest_HssVoiceDrv_2_01 | IA loopback Blocking: Maximum Voice Driver Clients (2 Analog, 1E1) | yes | Change to Use 2 Framers |
| systest_HssVoiceDrv_2_02 | IA loopback Blocking: Maximum Voice Driver Channels 2 Analog, 1 E1) | yes | Change to Use 2 Framers |
| systest_HssVoiceDrv_2_03 | IA loopback Non Blocking: Maximum Voice Driver Clients (2 Analog, 1E1) | no | this is a repeat of systest_HssVoiceDrv_2_01 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_2_04 | IA loopback Non-Blocking: Maximum Voice Driver Channels (2 Analog, 1 E1) | no | this is a repeat of systest_HssVoiceDrv_2_02 in a different driver mode which we are not compating s will not be run |
| systest_HssVoiceDrv_3_01 | IA loopback Channel enable/disable stress test: 128 channels (2 Analog, 1 E1) | yes | |

# V. Perl Script for BVA Test Code Generator

```
#!/usr/bin/perl

my @list;
my $i;
my $op;
my @param;
my $alreadyProcessed = "false";
my $bva = "";
my @min;
my @max;
my $paramCount = 0;
my $channelCount=0;

unlink "bvaValues.txt";
open (BVAVALUES, ">>bvaValues.txt");
close BVAVALUES;
my @processesParams; #list of inputs already BVA analysed

##extract test cases from BVA model
open (LOGFILE, "HSSVoiceNPEOnly.txt") or die "I couldn't get at the file";
for $line (<LOGFILE>)
{
        if($line =~ /operation/)
        {
                @list = split(' ', $line);
                $op = $list[1];
        }
        if($line =~ /input|output/)
        {
                my @param = split(' ', $line);
                open (BVAVALUES, "bvaValues.txt") or die "I couldn't get at the file";

                for $procLine(<BVAVALUES>)
                {
                        if($procLine =~ /$param[1]/)
                        {
                                $alreadyProcessed = "true";
                                last;
                        }
                }
         close BVAVALUES;
                if($alreadyProcessed eq "false")
                {
                        open (BVAVALUES, ">>bvaValues.txt");
                        print BVAVALUES "$op ";
            my $numParams = @param;
            for($i=1; $i<$numParams;$i++)
            {
                print BVAVALUES " $param[$i]";
            }
            print BVAVALUES "\n";
                       $paramCount++;
            close BVAVALUES;
               }
               $alreadyProcessed = "false";
        }
}
close LOGFILE;

my $testCount = 0;
my @testInputs;
#open (TEST_DATA, ">unchecked_test_case_data.txt ");
##GENERATE TEST CASES
for(my $i=0; $i<$paramCount; $i++)
{
        my @others; #list of other parameters in sytem not BVA analyed
        my $processedItem = 0;
        my $processCheck = "true";
        my $index = 0;
        my $paramBvaAnalysed = "false";
        open (BVAVALUES, "bvaValues.txt");
        for $line(<BVAVALUES>)
        {
               my @param = split(' ', $line);
        ##check list of parameters that we have processed
        foreach(@processesParams)
               {
                       if($_ eq $param[1])
                       {
                #set processed flag to true if current param is in the list
                $paramBvaAnalysed = "true";
                              last;
                       }
                       else
                       {
                              $paramBvaAnalysed = "false";
                       }
               }
               if(($paramBvaAnalysed eq "false") && ($processCheck eq "true"))
               {
                       $processCheck = "false";
                       $processedItem = $index;
            ##add current param to processes list so that we can skip it
            ##if we see it again
            push(@processesParams, $param[1]);
            ##save the minimum and maximum values for this parameter
            my $lengthOfParams = @param;
            for($j=0; $j<=($lengthOfParams-4);$j+=2)
            {
```

```
           push(@min, $param[$j+2]);
                          push(@max, $param[$j+3]);
        }
           }
           else
           {
        ##this parameter has already been processed to just use
        ##the minmium range value as default input
        push(@others, $param[2]);
           }
           $index++;
    }
    close BVAVALUES;

##colate the input values
my $numberOfRanges = @min;
for($j=0; $j<$numberOfRanges; $j++)
{
    my $counter=0;
           my $paramList1 = ""; #list of inputs against min BVA value
           my $paramList2 = ""; #list of inputs against max BVA value
    foreach(@others)
    {
           if($counter == $processedItem)
                   {
            $paramList1 = $paramList1.pop(@min)."\t".$_."\t";
                          $paramList2 = $paramList2.pop(@max)."\t".$_."\t";
                   }
                   else
                   {
                          $paramList1 = $paramList1.$_."\t";
                          $paramList2 = $paramList2.$_."\t";
                   }
           $counter++;
    }
    if($counter == $processedItem)
    {
    $paramList1 = $paramList1.pop(@min)."\t";
           $paramList2 = $paramList2.pop(@max)."\t";
    my $temp1 = 1;
    my $temp2 = 1;
    foreach (@testInputs)
    {
           if($_ eq $paramList1)
        {
            print "Removing duplicate input\n";
            $temp1 = 0;
        }
           if($_ eq $paramList2)
        {
            print "Removing duplicate input\n";
```

```
               $temp2 = 0;
            }
        }
        if($temp1)
        {
               push(@testInputs, $paramList1);
        }
        if($temp2)
        {
               push(@testInputs, $paramList2);
        }
        #print TEST_DATA "$paramList1\n";
               #print TEST_DATA "$paramList2";
       }
    else
    {
        my $temp1 = 1;
        my $temp2 = 1;
        foreach (@testInputs)
        {
               if($_ eq $paramList1)
            {
                print "Removing duplicate input\n";
               $temp1 = 0;
            }
               if($_ eq $paramList2)
            {
                print "Removing duplicate input\n";
               $temp2 = 0;
            }
        }
        if($temp1)
        {
               push(@testInputs, $paramList1);
        }
        if($temp2)
        {
               push(@testInputs, $paramList2);
        }
       #print TEST_DATA "$paramList1\n";
               #print TEST_DATA "$paramList2\n";
       }
    }
}
#print TEST_DATA "\n";
#close TEST_DATA;

##find and remove any duplicate test cases
open (TEST_DATA, ">test_case_data.txt ");
foreach (@testInputs)
{
        print TEST_DATA $_."\n";
}
close TEST_DATA;
##TURN TEST CASES INTO CODE
open (TEST_DATA, ">bva_test_data.c ");
open (TEST_CASES, ">temp.txt");
print TEST_CASES "systestHssDrvDesc_t s_testcaseList[] =\n{\n";
print TEST_DATA "
#ifdef __linux
#include <stdint.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#endif /* endif __linux */
#include \"icp_hssdrv.h\"
#include \"icp_hssvoicedrv.h\"
#include \"icp.h\"
#include \"IaHssDrvSystest.h\"
#include \"IaT1E1FramerSystest.h\"

uint32_t s_configure_framer = 1;

systestT1E1PortConfig_t portConfig =
        {ICP_FRAMERDRV_CFG_E1_CCS_HDB3_CRCMF_QUAD, ICP_FRAMERDRV_LOOPBACK_NONE,
                0, TRUE};

systestT1E1testData_t icp_T1E1SystestData[] =
{
        {1, {&portConfig , &portConfig, &portConfig}}
};\n\n";

my @testCaseData;
my $numbertestCases = 1;
open (TEST_CASE_DATA, "test_case_data.txt");
for $line (<TEST_CASE_DATA>)
{
        @testCaseData = split('\t', $line);
    $channelCount=0;
    my $channelsUsed = $testCaseData[5];

        my $testCase = $numbertestCases++;
        my $portNum = $testCaseData[1];
        my @portConfig = ("PORT_UNUSED", "PORT_UNUSED", "PORT_UNUSED", "PORT_UNUSED");
        my @portTsMap = ("0", "0", "0", "0");
        my @ChannelsOnPort= ("0", "0", "0", "0");
        $portConfig[$portNum] = "Q_E1_CGF_FLB";
        my $numClients = $testCaseData[0];
    my @tsMap = split(',',$testCaseData[3]);

        my @tsUsed = (0, 0, 0, 0);
        #my @timeSlotsRequiredToBeUsed = ("0","0","0");
        my @timeSlotsRequiredToBeUsed = split(',',$testCaseData[4]);
    #$timeSlotsRequiredToBeUsed[0] = $testCaseData[3];
    for($j=0; $j<4; $j++)
    {
        $timeSlotsRequiredToBeUsed[$j] =  hex($timeSlotsRequiredToBeUsed[$j]);
                $tsMap[$j] =  hex($tsMap[$j]);
    }
        my $channelsThisPort = 0;
    my $line;
    for($j=0; $j<4; $j++)
    {
        if($tsMap[$j]>0)
         {
                $line=$j;
             last;
         }
    }
    $portTsMap[$portNum]= "tc".$testCase."_".$portNum;
        print   TEST_DATA   "iaSystestHssDrvChanCfg_t   tc".$testCase."_".$portNum."\[\]
=\n{\n";
        for ($i=0; $i<$numClients; $i++)
        {
         $tsUsed[$line] =  $tsUsed[$line] | $tsMap[$line];
         addChannel(\@tsMap,$i);
                $ChannelsOnPort[$portNum]++;
         ##if we have more channels to add on this line
         if(       (($tsUsed[$line]       &       $timeSlotsRequiredToBeUsed[$line])!=
$timeSlotsRequiredToBeUsed[$line])
             ||($tsMap[$line] < 0x80000000) )
                {
                        $tsMap[$line] = $tsMap[$line]<<1;
                }
                else
                {
                        #$channelsThisPort = 0;
                        $tsMap[$line]=0;
             $line++;
                        if( $line >3)
                        {
                                $line = 0;

                        }
                        #$portConfig[$portNum] = "NPE_LOOPBACK";
                        $tsMap[$line] = 2;
                }
         if($ChannelsOnPort[$portNum] == 124)
         {
                if($portNum == 2)
             {
                $portNum=0;
             }
```

```
            else
            {
               $portNum++;
            }
            $portTsMap[$portNum]= "tc".$testCase."_".$portNum;
            $portConfig[$portNum] = "Q_E1_CGF_FLB";
            print TEST_DATA "\n};\n\n";
            print TEST_DATA "iaSystestHssDrvChanCfg_t tc".$testCase."_".$portNum."\[\]
=\n{\n";
        }
        ##if we have added all the channels to the clients check that we have
        ## fill all the required timeslots
        if( $i == ($numClients -1))
               {
            $tsMap[$line]=0;
            for($m=0; $m<4; $m++)
            {
                                     if($tsMap[$m]  >  0x80000000  ||  $tsMap[$m]  ==
0x00000000)
                                     {
                                             $tsMap[$m]=2;
                        while(($tsUsed[$m] & $tsMap[$m])
                              && $tsMap[$m] <= 0x80000000)
                        {
                              $tsMap[$m] = $tsMap[$m]<<1;
                        }
                                     }
               ##check that we have added the require number of Timeslots to this port
                              while(           (((~$tsUsed[$m]           )           &
$timeSlotsRequiredToBeUsed[$m]) > 0)
                      || $channelCount<$channelsUsed)
                              {
                    addChannel(\@tsMap,$i);
                    $ChannelsOnPort[$portNum]++;
        if($ChannelsOnPort[$portNum] == 124)
        {
               if($portNum == 2)
            {
               $portNum=0;
            }
            else
            {
               $portNum++;
            }
            $portTsMap[$portNum]= "tc".$testCase."_".$portNum;
            $portConfig[$portNum] = "Q_E1_CGF_FLB";
            print TEST_DATA "\n};\n\n";
            print TEST_DATA "iaSystestHssDrvChanCfg_t tc".$testCase."_".$portNum."\[\]
=\n{\n";
        }
                                     $tsUsed[$m] = $tsUsed[$m] | $tsMap[$m];
                    if(    (($tsUsed[$line]    &    $timeSlotsRequiredToBeUsed[$line])!=
$timeSlotsRequiredToBeUsed[$line])
                                ||($tsMap[$line] < 0x80000000) )
                                        {
                                                $tsMap[$line] = $tsMap[$line]<<1;
                                        }
                                        else
                                        {
                                                #$channelsThisPort = 0;
                                                $tsMap[$line]=0;
                                $line++;
                                                if( $line >3)
                                                {
                                                        $line = 0;
                                                }
                                                $tsMap[$line] = 2;
                                        }
                  }
                         }
                 }
        }
        print TEST_DATA "\n};\n\n";
        print TEST_CASES "\n\n";

        print TEST_CASES "\t{".$testCase.", ".$numClients.", 0,";
        print                                TEST_CASES                                  "
{".$portConfig[0].",".$portConfig[1].",".$portConfig[2].",".$portConfig[3]."},\n";
        print TEST_CASES "\t\t{".$portTsMap[0].", ".$portTsMap[1].", ".$portTsMap[2].",
0},";
        print    TEST_CASES    "    {".$ChannelsOnPort[0].",    ".$ChannelsOnPort[1].",
".$ChannelsOnPort[2].", 0},\n";
        print      TEST_CASES      "\t\tBLOCKING,      QOS_DISABLED,      SRTP_DISABLED,
QOS_PRIORITY_BOUNDARY_IS_0\n\t},";
}
print   TEST_CASES   "\n\t{999,   0,   0,   {PORT_UNUSED,   PORT_UNUSED,   PORT_UNUSED,
PORT_UNUSED},\n";
print TEST_CASES "\t\t{0,0,0,0}, {0,0,0, 0},\n";
print       TEST_CASES       "\t\tBLOCKING,       QOS_DISABLED,       SRTP_DISABLED,
QOS_PRIORITY_BOUNDARY_IS_0\n\t}";
print TEST_CASES "\n};\n";
close TEST_CASE_DATA;
close TEST_DATA;
close TEST_CASES;
open (TEST_DATA, ">>bva_test_data.c ");
open (TEST_CASES, "temp.txt");
for $line (<TEST_CASES>)
{
        print TEST_DATA $line;
}
close TEST_DATA;
close TEST_CASES;
```

```
unlink "temp.txt";
#unlink "test_case_data.txt";

sub addChannel
{
        (my $reftsMap, $i) = @_;
    my @tsMap = @{$reftsMap};
    my $map="{";
    my $chan_cfg = "";
    for($n=0; $n<4; $n++)
    {
        if($tsMap[$n]==0)
        {
         }
         elsif($tsMap[$n]%15==0)
         {
                $chan_cfg = "CHAN_CFG_320B";
         }
         elsif($tsMap[$n]%3==0)
         {
                $chan_cfg = "CHAN_CFG_160B";
         }
         else
         {
                $chan_cfg = "CHAN_CFG_DFT";
         }
         $tsMap[$n] = sprintf("0x%08x", $tsMap[$n]);
         $map = $map.$tsMap[$n];
         if($n!=3)
         {
                $map= $map.",";
         }
    }
    $map = $map."}";
    if($channelCount!= 0 && $channelCount!= 124)
    {
        print TEST_DATA ",\n";
    }
    print TEST_DATA "\t{".$i.", ".$map.", ".$chan_cfg.", VOICE_CHANNEL, ".$i."}";
        for($n=0; $n<4; $n++)
        {
                $tsMap[$n] =  hex($tsMap[$n]);
        }
    $channelCount++;
}
```

# VI. Perl Script for DU Test Case Generator

```
#!/usr/bin/perl


my @globalDefs;
my @list;
my $line;
#array to store pre-post conditions of an operation
my @operationtestCode = (0,0,0,0,0,0,0,0,0,0);

##Part 1: open specification file and extract global definitions
open (RESOLVE_SPEC, "HSSVoiceResolveSpec.txt") or die "I couldn't get at the file";
for $line (<RESOLVE_SPEC>)
{
	if($line !~ /;/)
	{
		if($line =~ /\t/)
		{
			@list = split(':', $line);
			$list[0] =~ s/\t//;
			push(@globalDefs, $list[0]);
		}
	}
	else
	{
		last;
	}
}
close RESOLVE_SPEC;
##declare some memory to help create DU test code
#list & normal order of function calls for API under test
my @allFunctions = ("init", "open", "portUp", , "channelAdd", , "channelUp",
 "read", "write", "channelDown", "channelRemove", "portDown", "close");
my $testCode = "";
my $testCaseCount  = 0;
my $counter = 0;
my $currentOp ="";	#used to store operations being processed in RESOLVE_SPEC
my $funcParams = "";	#used to store parameters to operation be processed
my $initCode = "";		#used to store the initilisation code of global vars
my $resetCode = "";		#used to store the initilisation code of global vars
my $tsMapCode = "";

open (DU, ">duCoverage.txt") or die "I couldn't get at the file";
open (TESTCODE, ">duPsuedoCode.c") or die "I couldn't get at the file";

## Part2: for each global definition, find valid DU pairs and create test code for
## each pair
foreach(@globalDefs)
{
    print DU "\n\n\n***** DU Coverage for $_ *****\n";
    print DU "testCase Defined Used\n";
    open (RESOLVE_SPEC, "HSSVoiceResolveSpec.txt") or die "I couldn't get
           at the file";
    my $lineNum = 1;  #track current line be processed in RESOLVE_SPEC
    my @defined;      #store lineNo's where current globalDef is defined
    my @used;         #store lineNo's where current globalDef is used
my @useCaseFunctionCall;    #store operation&parameters of use
my @defineCaseFunctionCall; #store operation&parameters of define
    my $processingRequires = "false";  #flag set id requires spans >1 line
    my $processingEnsures = "false";   #flag set id ensures spans >1 line
    my $functionsWhichDefine = "";   #string of functions which define is used
my $timeSlotSetting = "";
for $line (<RESOLVE_SPEC>)
    {
    #if we find a new operation
    if($line =~ /^operation/)
    {
        @list = split(' ', $line);
        #store the operation name
        $currentOp = $list[1];
        #reset the funcParams (following consumes lines define params for
        #this operation)
        $funcParams = "";
                   $processingRequires = "false";
                   $processingEnsures = "false";
    }
    #if we find a line with consumes
    if($line =~ /^\tconsumes/)
    {
           @list = split(' ', $line);
        #add it to the function params for current operation being processed
        $funcParams = $funcParams." ".$list[1];
    }
    #use case is found, or flag is set indicating we are processing a use
           #
    if($line =~ /^requires(.*|\W*)$_/ || $processingRequires eq $_)
           {
        my $useInput="";
        @list = split(' ', $line);
        if($processingRequires eq $_)
        {
           $list[0] =~ s/\t//;
            my $var = $list[0];
            $var =~ s/\W.+//;
            if($var =~ /tsMap/)
            {
                  $useInput=$list[0].";\n";
            }
        }
        else
```

```
            {
               $list[1] =~ s/\n//;
               my $var = $list[1];
               $var =~ s/\W.+//;
               if($var =~ /tsMap/)
               {
                              $useInput=$list[1].";\n";
               }
            }

            if( ($line =~ /tsMap\.line0_timeslot_bit_map==/) ||
               ($line =~ /tsMap\.line1_timeslot_bit_map==/) ||
                       ($line =~ /tsMap\.line2_timeslot_bit_map==/) ||
                       ($line =~ /tsMap\.line3_timeslot_bit_map==/))
               {
                       @tsMapConfig = split('&&',$line);
               $tsMapConfig[0] =~ s/requires: \(//;
               $tsMapConfig[0] =~ s/==/=/;
               $tsMapConfig[0] =~ s/ |\(//g;
               $tsMapConfig[1] =~ s/==/=/;
               $tsMapConfig[1] =~ s/ |\(//g;
               $tsMapConfig[2] =~ s/==/=/;
               $tsMapConfig[2] =~ s/ |\(//g;
               $tsMapConfig[3] =~ s/==/=/;
               $tsMapConfig[3] =~ s/ |\(|\)//g;
               $tsMapConfig[3] =~ s/\).+ |\|\|.+//;
               chomp($tsMapConfig[3]);
               $tsMapCode                                                =
$tsMapConfig[0].";\n".$tsMapConfig[1].";\n".$tsMapConfig[2].";\n".$tsMapConfig[3].";\n
"
               }
               else
               {
               }
            #store the lineNumber where a globalDef is used
            push (@used,$lineNum);
            #store the function call and parameters for this use
            push
(@useCaseFunctionCall,"$tsMapCode\n$useInput$currentOp($funcParams)");
            #set processing flag to the use we are processing
            $processingRequires = $_;

               }
        #define case is found, or flag is set indicating we are processing a def
               if($line =~ /^ensures\W+\(*$_/ || $processingEnsures eq $_)
               {
            #set processing flag to the define we are processing
            $processingEnsures = $_;
                       #store the lineNumber where a globalDef is defined
            push (@defined, $lineNum);
            #store the functions that this global variable is defined in
            if($functionsWhichDefine !~ /$currentOp/)
            {
               $functionsWhichDefine = $functionsWhichDefine.$currentOp." ";
            }
            ##RULE if we are defining timeslot being set on a spefific line then
            if($line  =~  /tsUsed.line\d_timeslot_bit_map/  &&  $processingEnsures  eq
"tsUsed")
            {
                print "$&\n";
                my $tsStr = $&;
                $tsStr =~ s/tsUsed/tsMap/g;
                $timeSlotSetting                                                 =
"tsMap.line0_timeslot_bit_map=0;\n".$tsStr."=2;\n\n";
            }
            ##RULE  Should call all functions before a define
            #get the functions that can be called for the current operation
            # where we have found a define and create the function calls for it
            foreach $func (@allFunctions)
            {
                #if the function is not the current operation insert the funtion
                #call into the code
                if($func ne $currentOp)
                {
                      $testCode = $testCode."$func\n";
                }
                else
                {
                    #insert the current operation and its parameters into the
                    #test code
                    $testCode = $testCode.$timeSlotSetting."$currentOp($funcParams)";
                    $timeSlotSetting = "";
                    #then store it
                    push (@defineCaseFunctionCall,$testCode);
                    #reset the test code memory
                    $testCode = "";
                    #escape foreach loop
                    last;
                }
            }
               }
        #if the line is a global definition
               if($line =~ /^\t$_/)
               {
            @list = split(' ', $line);
            if(@list ==2)
            {
                              $initCode = $initCode.$list[1]." ".$_.";\n";
                #$resetCode = $resetCode.$_." = 0;\n\t";
                $resetCode = $resetCode."\tbzero(&".$_.", sizeof(".$list[1]."));\n";
            }
            elsif(@list==3)
            {
                my $arraySize = $list[2];
                $arraySize =~ s/[\[|\]]//g;
                #print "ArraySize for $_ : $arraySize\n";
                $initCode = $initCode.$list[1]." ".$_.$list[2].";\n";
                $resetCode                 =                $resetCode."\tbzero(".$_.",
sizeof(".$list[1].")*".$arraySize.");\n";
                #print $resetCode;
            }
            #store is as a definition
            push (@defined, $lineNum);
            #TODO REMOVE THE FOLLOWING LINE
            push (@defineCaseFunctionCall,"init");
               }
        #if there is no '\' in the line, then we have finsihed processing the
        #current define or use.
        if($line !~ /\\/ )
               {
                       #reset the flags
            $processingRequires = "false";
                       $processingEnsures = "false";
               }
        #increment the line number that we are looking at in RESOLVE_SPEC
               $lineNum++;
        }
    close RESOLVE_SPEC;
    #processing of 1 globalDef complete, now output the valid DU pairs and test
    #code for it

    #for each define
        foreach $def (@defined)
        {
               foreach(@used)
               {
            #increment the testcase count
            $testCaseCount++;
            #output a DU pair
            print DU "$testCaseCount $def $_\n";
               }
        }
    print DU "\n";
    my $lengthUseCase = @useCaseFunctionCall;
    my $lengthDefineCase = @defineCaseFunctionCall;
    print TESTCODE "/******DU testcases for $_******/\n";
        for ($k=0; $k<$lengthDefineCase; $k++)
    {
               for ($j=0; $j<$lengthUseCase; $j++)
               {
                       $counter++;
            ##RULE Should only call functions after a definition that dont
            ##re-define before use
            ##this section of code determines functions that maybe called
            ##after, the function that causes definition, that do not redefine
            ##the variable we are analysing
            my $index =0;
            #find the index of the useCase function from the list of all
            #functions
            my $usetestCode = $useCaseFunctionCall[$j];
            foreach $func (@allFunctions)
            {
               if($usetestCode =~ /$func/)
                {
                    $index--;
                    last;
                }
                else
                {
                       $index++;
                }
            }
            ##search back in the list of all functions from the useCase Function
            ## and add functions which do not define and are not in the list of
            ##functions at or before the defined function
            my $dependendiesMet = 1; #flag set if
            while ($defineCaseFunctionCall[$k] !~ /$allFunctions[$index]/ &&
               ($index > 0))
            {
               if( ($functionsWhichDefine !~ /$allFunctions[$index]/) &&
                               ($defineCaseFunctionCall[$k]  !~ /$allFunctions[$index]/)
)
                {
                    $dependendiesMet = &AllDependenciesAvailible(

        $allFunctions[$index],

        $functionsWhichDefine,
                                             $defineCaseFunctionCall[$k]);
                    if($dependendiesMet)
                    {
                              $usetestCode = $allFunctions[$index]."\n$usetestCode";
                    }
                }
                $index--;
            }
            ##RULE where close is called in succession is not a valid
            #test case
            ##RULE if dependcies cannot be meet it is not a valid test case

#            || &AllDependenciesAvailible($useCaseFunctionCall[$j],
#
        $functionsWhichDefine,
#                                       $usetestCode) == 0)
            #
            if(($defineCaseFunctionCall[$k] =~ /close\( fileDesc\)$/ &&
               $usetestCode =~ /^close\( fileDesc\)/) || $dependendiesMet ==0)

            {
               print "test Case $counter Exluded from psuedo code\n";
                print "DefineCase Function Call: $defineCaseFunctionCall[$k]\n\n";
                print "UseCase testCode: $usetestCode\n\n";
                print "dependendies meet: $dependendiesMet\n\n\n\n";
            }
               else
            {
                #if($dependendiesMet)
                              print TESTCODE "test Case $counter(void)\n{\n";
               print TESTCODE "$defineCaseFunctionCall[$k]\n$usetestCode\n}\n\n";
            }
               }
        }
        print DU "\n";
    print TESTCODE "\n\n";

}
close DU;
close TESTCODE;
print "test case count $testCaseCount\n";


##PART3

#open the RESOLVE_SPEC again and extract out all the pre and post conditions
#for each opearion and create test code to check and set them
open (RESOLVE_SPEC, "HSSVoiceResolveSpec.txt") or die "I couldn't get
               at the file";
my $processingRequires = "false";  #flag set id requires spans >1 line
my $processingEnsures = "false";   #flag set id ensures spans >1 line
my $preConditions ="\tif("; #store the precondition of an operation to this
my $postConditions = "\t"; #store the post condition code of an operation
my $braceOpen = "false";
my @tsMapConfig;
for $line (<RESOLVE_SPEC>)
{
    #if we find a new operation
    if($line =~ /^operation/)
    {
        #if we were processing a previos Operation close the postConditions
        if($postConditions ne "")
        {
            $postConditions=$postConditions."}\n\telse\n\t{\n";
            $postConditions=$postConditions."\t\tprintf     (\"$currentOp:     Setting
ResultExpected = -1\\n\");\n";
```

```
            $postConditions=$postConditions."\t\tResultExpected = -1;\n\t}\n";
        }
        #store the test code
        if($currentOp eq "open")
        {
               $operationtestCode[0] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "portUp")
        {
               $operationtestCode[1] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "channelAdd")
        {
               $operationtestCode[2]                                                =
$preConditions.$postConditions."\ttsMap.line0_timeslot_bit_map                      =
tsMap.line0_timeslot_bit_map<<1;\n";
        }
        elsif($currentOp eq "channelUp")
        {
               $operationtestCode[3] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "read")
        {
               $operationtestCode[4] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "write")
        {
               $operationtestCode[5] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "channelDown")
        {
               $operationtestCode[6] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "channelRemove")
        {
               $operationtestCode[7] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "portDown")
        {
               $operationtestCode[8] = $preConditions.$postConditions;
        }
        elsif($currentOp eq "close")
        {
               $operationtestCode[9] = $preConditions.$postConditions;
        }
        #reset the pre& post conditions for new operation
        $preConditions ="\tif(";
       $postConditions = "\t";

        @list = split(' ', $line);
        #store the operation name
```

```
    $currentOp = $list[1];
}
#(.*|\W*)$_
if($line =~ /^requires/ || $processingRequires eq "true")
    {
    if( ($line !~ /tsMap\.line0_timeslot_bit_map==/) &&
                   ($line !~ /tsMap\.line1_timeslot_bit_map==/) &&
                   ($line !~ /tsMap\.line2_timeslot_bit_map==/) &&
                   ($line !~ /tsMap\.line3_timeslot_bit_map==/))
    {
           if($processingRequires eq "true")
        {
            my $temp = $line;
            $temp =~ s/\t//g;
            $temp =~ s/\\//;
            $temp =~ s/\n//;
            $preConditions = $preConditions.$temp;

            if(($line !~ /\\/) && $braceOpen eq "true")
            {
                   $braceOpen = "false";
                   $preConditions = $preConditions.")";
            }
        }
        else
        {
           @list = split(':', $line);
           #remove and possible backslash charater
                   $list[1] =~ s/\\//;
            $list[1] =~ s/\n//;
            $list[1] =~ s/^\s//;
           #add an && statement to preconditions if there is an existing
           #condtion
           if($preConditions !~ /if\($/ && $braceOpen eq "false")
           {
                   $preConditions = $preConditions." &&\n\t\t ";
           }
            ## if the statement is over multiple lines open (
            if($line =~/\\/)
            {
                           if($braceOpen eq "false")
                    {
                           $preConditions = $preConditions."(";
                           $braceOpen = "true";
                    }
            }
            #store the requriements of operation
            $preConditions = $preConditions.$list[1];
           #$preConditions = $preConditions." ";
           }
        #set processing flag to the use we are processing
```

```
            $processingRequires = "true";
        }
        }
    #define case is found, or flag is set indicating we are processing a def
    #\W+\(*$_
    if($line =~ /^ensures/ || $processingEnsures eq "true")
        {
        #get the post condition value
        @list = split(':', $line);
        $list[1] =~ s/\n//;
        $list[1] =~ s/^\s//;
        #add it to the post condition code
        $postConditions =  $postConditions."\t$list[1];\n\t";
        #set processing flag to the define we are processing
        $processingEnsures = "true";
        }
    #if there is no '\' in the line, then we have finsihed processing the
    #current define or use.
    if($line !~ /\\/ )
        {
                #reset the flags
        $processingRequires = "false";
                $processingEnsures = "false";
        }
    #if we have gone to the end of the pre-conditions close the
    #pre conditions statement
    if($line =~ /#postconditions/)
    {
        $preConditions = $preConditions.")\n\t{\n\t";
        $postConditions = "\tprintf (\"$currentOp: ResultExpected = 0\\n\");\n\t";
        $postConditions = $postConditions."\tResultExpected = 0;\n\t";
    }
}
$postConditions=$postConditions."}\n\telse\n\t{\n";
$postConditions=$postConditions."\t\tprintf (\"$currentOp: Setting ResultExpected = -
1\\n\");\n";
$postConditions=$postConditions."\t\tResultExpected = -1;\n\t}\n";
if($currentOp eq "close")
{
        $operationtestCode[9] = $preConditions.$postConditions;
}

#PART4
##REPLACE PSUEDO CODE WITH REAL CODE
open (PSUEDOCODE, "duPsuedoCode.c") or die "I couldn't get at the file";
open (TESTCODE, ">dutestCode.c") or die "I couldn't get at the file";
open (HEADERCODE, ">dutestCode.h") or die "I couldn't get at the file";

print HEADERCODE "#include <asm/types.h>\n";

print TESTCODE "#include <asm/types.h>\n";
print TESTCODE "#include <sys/stat.h>\n";
print TESTCODE "#include <fcntl.h>\n";
print TESTCODE "#include <sys/ioctl.h>\n";
print TESTCODE "#include <unistd.h>\n";
print TESTCODE "#include <string.h>\n";
print TESTCODE "#include \"icp.h\"\n";
print TESTCODE "#include \"readWrite.h\"\n";
print TESTCODE "#include \"icp_hssdrv.h\"\n\n";
print TESTCODE "#include \"icp_hssvoicedrv.h\"\n\n";
print TESTCODE "#include \"dutestCode.h\"\n\n";
$initCode = $initCode."\n\n\n";
$initCode = $initCode."#define PASS 0\n";
$initCode = $initCode."#define FAIL 1\n";
$initCode = $initCode."#define NOT_SET 0\n";
$initCode = $initCode."#define DOWN 0\n";
$initCode = $initCode."#define UP 1\n\n";
$initCode = $initCode."int i;\n";
$initCode = $initCode."int j;\n";
$initCode = $initCode."int ResultExpected;\n";
$initCode = $initCode."int fd[128];\n";
$initCode = $initCode."int portNum;\n";
$initCode = $initCode."int channelNum;\n";
$initCode = $initCode."icp_hssdrv_portup_t pCfg;\n";
$initCode = $initCode."icp_hssvoicedrv_channeladd_t cCfg;\n";
$initCode = $initCode."int config;\n";
print TESTCODE $initCode."\n\n";

print TESTCODE "void init()\n{\n";
print TESTCODE "\tbzero(fd, sizeof(int)*128);\n";
print TESTCODE $resetCode;
print TESTCODE "\tResultExpected=0;\n";
print TESTCODE "\tportNum=0;\n";
print TESTCODE "\tchannelNum=0;\n";
print TESTCODE "\tpCfg.portId=0;\n";
print TESTCODE "\tpCfg.port_config=ICP_HSSDRV_PORT_HMVIP_FRAMER_MEZZANINE_CONFIG;\n";
print TESTCODE "\tpCfg.loopbackMode=ICP_HSSDRV_NO_LOOPBACK;\n";
print TESTCODE "\tcCfg.channelId = channelNum;\n";
print TESTCODE "\tcCfg.portId = portNum;\n";
print TESTCODE "\tcCfg.voicePacketSize = 240;\n";
print TESTCODE "\ttsMap.line0_timeslot_bit_map = 2;\n";
print TESTCODE "\tcCfg.tsMap.line0_timeslot_bit_map = 2;\n";
print TESTCODE "\tcCfg.tsMap.line1_timeslot_bit_map = 0;\n";
print TESTCODE "\tcCfg.tsMap.line2_timeslot_bit_map = 0;\n";
print TESTCODE "\tcCfg.tsMap.line3_timeslot_bit_map = 0;\n";
print TESTCODE "\tcCfg.voiceIdleAction = 0;\n";
print TESTCODE "\tcCfg.voiceIdlePattern = 0x7E;\n";
print TESTCODE "\tcCfg.channelDataInvert =0;\n";
print TESTCODE "\tcCfg.channelBitEndianness = 1;\n";
print TESTCODE "\tcCfg.channelByteSwap = 0;\n";

print TESTCODE "\tconfig=ICP_HSSDRV_PORT_HMVIP_FRAMER_MEZZANINE_CONFIG;\n";
print TESTCODE "\tfor(i=0;i<3;i++)\n\t{\n";
print TESTCODE "\t\ticp_FramerDrvInit(i);\n";
print TESTCODE "\t\ticp_FramerDrvConfigSet(i,ICP_FRAMERDRV_CFG_E1_CCS_HDB3_DATA);\n";
print TESTCODE "\t\ticp_FramerDrvLoopbackSet(i,ICP_FRAMERDRV_LOOPBACK_DIGITAL);\n";
print TESTCODE "\t}\n";
print TESTCODE "}\n";
print TESTCODE "void uninit()\n{\n";
print TESTCODE &channelDownCode();
print TESTCODE &channelRemoveCode();
print TESTCODE &portDownCode();
print TESTCODE &closeCode();
print TESTCODE "\tfor(i=0;i<3;i++)\n\t{\n";
print TESTCODE "\t\ticp_FramerDrvUninit(i);\n";
print TESTCODE "\t}\n";
print TESTCODE "}\n";

for $line (<PSUEDOCODE>)
{
    if($line =~ /test Case/)
    {
       $line =~ s/test Case /uint8_t test/;
        print TESTCODE $line;
        my $proto = $line;
        $proto =~ s/\n/;\n/;
        print HEADERCODE $proto;
#        $line =~ s/\*\/$//;
    }
    elsif($line =~ /^init/)
    {
       print TESTCODE "\tinit();\n";
    }
    elsif($line =~ /^open/)
    {
       print TESTCODE $operationtestCode[0];
        print TESTCODE "\tfd[openCounter] = open(\"/dev/hss-voice\",O_RDWR);\n";
        print TESTCODE "\tif(fd[openCounter] < ResultExpected)\n\t{\n";
        print TESTCODE "\t\tuninit();\n";
        print TESTCODE "\t\treturn FAIL;\n\t}\n";
    }
    elsif($line =~ /^portUp/)
    {
       print TESTCODE $operationtestCode[1];
        print TESTCODE "\tif(ioctl(fd[openCounter], ICP_HSSVOICEDRV_PORT_UP, &pCfg)";
        &failCode();

    }
    elsif($line =~ /^channelAdd/)
    {
        print TESTCODE "\tcCfg.channelId = ++channelNum;\n";
        print TESTCODE "\tcCfg.portId = portNum;\n";
        print TESTCODE "\tcCfg.voicePacketSize = 240;\n";
        print          TESTCODE         "\tcCfg.tsMap.line0_timeslot_bit_map          =
tsMap.line0_timeslot_bit_map;\n";

        print          TESTCODE         "\tcCfg.tsMap.line1_timeslot_bit_map          =
tsMap.line1_timeslot_bit_map;\n";
        print          TESTCODE         "\tcCfg.tsMap.line2_timeslot_bit_map          =
tsMap.line2_timeslot_bit_map;\n";
        print          TESTCODE         "\tcCfg.tsMap.line3_timeslot_bit_map          =
tsMap.line3_timeslot_bit_map;\n";
        print TESTCODE "\tcCfg.voiceIdleAction = 0;\n";
        print TESTCODE "\tcCfg.voiceIdlePattern = 0x7E;\n";
        print TESTCODE "\tcCfg.channelDataInvert =0;\n";
        print TESTCODE "\tcCfg.channelBitEndianness = 1;\n";
        print TESTCODE "\tcCfg.channelByteSwap = 0;\n";
        print TESTCODE $operationtestCode[2];
        print    TESTCODE    "\ti=    ioctl(fd[openCounter],    ICP_HSSVOICEDRV_CHAN_ADD,
&cCfg);\n";
               print TESTCODE "\tif(i!=0) channelNum--;\n";
        print TESTCODE "\tif(i";
        &failCode();
    }
    elsif($line =~ /^channelUp/)
    {
       print TESTCODE $operationtestCode[3];
        print    TESTCODE    "\ti=    ioctl(fd[openCounter],    ICP_HSSVOICEDRV_CHAN_UP,
channelNum);\n";
        print TESTCODE "\tif(i";
        &failCode();
    }
    elsif($line =~ /^read/)
    {
       print TESTCODE "\tif(systestHssDrvVoiceTx (openCounter, 10, 240)!=0)\n\t{\n";
        print TESTCODE "\t\tuninit();\n";
        print TESTCODE "\t\treturn FAIL;\n\t}\n";
    }
    elsif($line =~ /^write/)
    {
    }
    elsif($line =~ /^channelDown/)
    {
       print TESTCODE $operationtestCode[6];
        print    TESTCODE    "\ti=    ioctl(fd[openCounter],    ICP_HSSVOICEDRV_CHAN_DOWN,
channelNum);\n";
        print TESTCODE "\tif(i";
        &failCode();
    }
    elsif($line =~ /^channelRemove/)
    {
       print TESTCODE $operationtestCode[7];
        print    TESTCODE    "\ti=    ioctl(fd[openCounter],    ICP_HSSVOICEDRV_CHAN_REMOVE,
channelNum);\n";
```

```
        print TESTCODE "\tif(i==0) channelNum--;\n";
               print TESTCODE "\tif(i";
        &failCode();
    }
    elsif($line =~ /^portDown/)
    {
       print TESTCODE $operationtestCode[8];
        print   TESTCODE   "\ti=   ioctl(fd[openCounter],   ICP_HSSVOICEDRV_PORT_DOWN,
portNum);\n";
#        print TESTCODE "\tif(i!=0) \n\t{\n";
#              print TESTCODE &channelRemoveCode();
#        print TESTCODE "\t}\n";
        print TESTCODE "\tif(i";
        &failCode();
    }
    elsif($line =~ /^close/)
    {
       print TESTCODE $operationtestCode[9];
        print TESTCODE "\ti= close(fd[openCounter]);\n";
        print TESTCODE "\tif(i";
        &failCode();
#        print TESTCODE "\t!= ResultExpected)\n{\n";
 #       print TESTCODE "\t\tuninit();\n";
#        print TESTCODE "\t\treturn FAIL;\n\t}\n";
    }
    elsif($line =~ /^}/)
    {
       print TESTCODE "\tuninit();\n\treturn PASS;\n}\n";
    }
    else
    {
       print TESTCODE $line;
    }
}
print HEADERCODE "\n";
close HEADERCODE;
close PSUEDOCODE;
close TESTCODE;




sub AllDependenciesAvailible
{
       my ($func, $functionsWhichDefine, $defineCaseFunctionCall) = @_;
    my $returnVal =0;
    if($defineCaseFunctionCall =~ /$func/)
    {
       return 1;
    }
    if($functionsWhichDefine =~ /$func/)
    {
       return 0;
    }
    if($func =~ /channelDown/)
    {
       $returnVal = &AllDependenciesAvailible("channelUp",
               $functionsWhichDefine, $defineCaseFunctionCall);
    }
    elsif($func =~ /read/)
    {
       $returnVal = &AllDependenciesAvailible("channelUp",
               $functionsWhichDefine, $defineCaseFunctionCall);
    }
    elsif($func =~ /write/)
    {
       $returnVal = &AllDependenciesAvailible("channelUp",
               $functionsWhichDefine, $defineCaseFunctionCall);
    }
    elsif($func =~ /channelUp/)
    {
       $returnVal = &AllDependenciesAvailible("channelAdd",
               $functionsWhichDefine, $defineCaseFunctionCall);
    }
    elsif($func =~ /channelAdd/)
    {
       $returnVal = &AllDependenciesAvailible("open",
               $functionsWhichDefine, $defineCaseFunctionCall);
    }
    elsif($func =~ /portUp/)
    {
       $returnVal = &AllDependenciesAvailible("open",$functionsWhichDefine,
               $defineCaseFunctionCall);
    }
    elsif($func =~ /portDown/)
    {
       $returnVal = &AllDependenciesAvailible("open",$functionsWhichDefine,
               $defineCaseFunctionCall);
    }
    elsif($func =~ /open/)
    {
      $returnVal = &AllDependenciesAvailible("init",$functionsWhichDefine,
               $defineCaseFunctionCall);
    }
    elsif($func =~ /close/)
    {
      $returnVal = &AllDependenciesAvailible("open",$functionsWhichDefine,
               $defineCaseFunctionCall);
    }
    return $returnVal;
}
```

```
sub closeCode
{
        my $str= "\tfor(i=0;i<openCounter;i++)\n\t{\n";
    $str = $str."\t\tclose(fd[i]);\n";
    $str = $str."\t}\n";
    return $str;
}

sub portDownCode
{
    my $str=  "\tfor(i=0;i<3;i++)\n\t{\n";
    $str = $str."\t\tif(portStatus[i] == UP)\n\t\t{\n";
    $str = $str."\t\t\tioctl(fd[openCounter], ICP_HSSVOICEDRV_PORT_DOWN, 0);\n";
    $str = $str."\t\t}\n";
    $str = $str."\t}\n";
        return $str;
}

sub channelRemoveCode
{
        my $str=  "\tfor(j=channelNum;j>0;j--)\n\t{\n";
    $str = $str."\t\tioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_REMOVE,j);\n";
    $str = $str."\t}\n";
        return $str;
}

sub channelDownCode
{
        my $str=  "\tfor(i=channelNum;i>0;i--)\n\t{\n";
    $str = $str."\t\tioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_DOWN,i);\n";
    $str = $str."\t}\n";
        return $str;
}

sub failCode
{
        print TESTCODE " != ResultExpected)\n\t{\n";
        print TESTCODE "\t\tprintf (\"Result: %d, was not expected\\n\", i);\n";
        print TESTCODE "\t\tuninit();\n";
        print TESTCODE "\t\treturn FAIL;\n\t}\n";
}
```

# VII. Sample of Test Code Generated by BVA Test Generator

```
#ifdef __linux
#include <stdint.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#endif /* endif __linux */

#include "icp_hssdrv.h"
#include "icp_hssvoicedrv.h"
#include "icp.h"
#include "IaHssDrvSystest.h"
#include "IaT1E1FramerSystest.h"


uint32_t s_configure_framer = 1;


systestT1E1PortConfig_t portConfig =
    {ICP_FRAMERDRV_CFG_E1_CCS_HDB3_CRCMF_QUAD, ICP_FRAMERDRV_LOOPBACK_NONE,
        0, TRUE};

systestT1E1testData_t icp_T1E1SystestData[] =
{
    {1, {&portConfig , &portConfig, &portConfig}}
};

iaSystestHssDrvChanCfg_t tc1_0[] =
{
        {0, {0x00000002,0x00000000,0x00000000,0x00000000}, CHAN_CFG_DFT, VOICE_CHANNEL,
0}
};


iaSystestHssDrvChanCfg_t tc3_2[] =
{
        {0, {0x00000002,0x00000000,0x00000000,0x00000000}, CHAN_CFG_DFT, VOICE_CHANNEL,
0}
};

iaSystestHssDrvChanCfg_t tc4_0[] =
{
        {0, {0x00000002,0x00000000,0x00000000,0x00000000}, CHAN_CFG_DFT, VOICE_CHANNEL,
0}
};


systestHssDrvDesc_t s_testcaseList[] =
{
        {1, 1, 0, {Q_E1_CGF_FLB,PORT_UNUSED,PORT_UNUSED,PORT_UNUSED},
               {tc1_0, 0, 0, 0}, {1, 0, 0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },

        {2, 128, 0, {Q_E1_CGF_FLB,Q_E1_CGF_FLB,PORT_UNUSED,PORT_UNUSED},
               {tc2_0, tc2_1, 0, 0}, {124, 4, 0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },

        {3, 1, 0, {PORT_UNUSED,PORT_UNUSED,Q_E1_CGF_FLB,PORT_UNUSED},
               {0, 0, tc3_2, 0}, {0, 0, 1, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },

        {4, 1, 0, {Q_E1_CGF_FLB,PORT_UNUSED,PORT_UNUSED,PORT_UNUSED},
               {tc4_0, 0, 0, 0}, {1, 0, 0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },


        {28, 1, 0, {Q_E1_CGF_FLB,Q_E1_CGF_FLB,PORT_UNUSED,PORT_UNUSED},
               {tc28_0, tc28_1, 0, 0}, {124, 0, 0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },

        {29, 1, 0, {Q_E1_CGF_FLB,Q_E1_CGF_FLB,PORT_UNUSED,PORT_UNUSED},
               {tc29_0, tc29_1, 0, 0}, {124, 4, 0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        },
        {999, 0, 0, {PORT_UNUSED, PORT_UNUSED, PORT_UNUSED, PORT_UNUSED},
               {0,0,0,0}, {0,0,0, 0},
               BLOCKING, QOS_DISABLED, SRTP_DISABLED, QOS_PRIORITY_BOUNDARY_IS_0
        }
};
```

# VIII. Sample of DU Generated Test Code

```
#include <asm/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <string.h>
#include "icp.h"
#include "readWrite.h"
#include "icp_hssdrv.h"

#include "icp_hssvoicedrv.h"

#include "dutestCode.h"

int portStatus[3];
int channelsOnPort[3];
int channelsOnFd[128];
int openCounter;
int channelsUsed;
icp_hssdrv_timeslot_map_t tsUsed;
int portConfig[3];
char channelStatus[128];
icp_hssdrv_timeslot_map_t channelConfig[128];
char readBuffer[40960];
char writeBuffer[40960];
icp_hssdrv_timeslot_map_t tsMap;



#define PASS 0
#define FAIL 1
#define NOT_SET 0
#define DOWN 0
#define UP 1

int i;
int j;
int ResultExpected;
int fd[128];
int portNum;
int channelNum;
icp_hssdrv_portup_t pCfg;
icp_hssvoicedrv_channeladd_t cCfg;
int config;


void init()
{
        bzero(fd, sizeof(int)*128);
        bzero(portStatus, sizeof(int)*3);
        bzero(channelsOnPort, sizeof(int)*3);
        bzero(channelsOnFd, sizeof(int)*128);
        bzero(&openCounter, sizeof(int));
        bzero(&channelsUsed, sizeof(int));
        bzero(&tsUsed, sizeof(icp_hssdrv_timeslot_map_t));
        bzero(portConfig, sizeof(int)*3);
        bzero(channelStatus, sizeof(char)*128);
        bzero(channelConfig, sizeof(icp_hssdrv_timeslot_map_t)*128);
        bzero(readBuffer, sizeof(char)*40960);
        bzero(writeBuffer, sizeof(char)*40960);
        bzero(&tsMap, sizeof(icp_hssdrv_timeslot_map_t));
        ResultExpected=0;
        portNum=0;
        channelNum=0;
        pCfg.portId=0;
        pCfg.port_config=ICP_HSSDRV_PORT_HMVIP_FRAMER_MEZZANINE_CONFIG;
        pCfg.loopbackMode=ICP_HSSDRV_NO_LOOPBACK;
        cCfg.channelId = channelNum;
        cCfg.portId = portNum;
        cCfg.voicePacketSize = 240;
        tsMap.line0_timeslot_bit_map = 2;
        cCfg.tsMap.line0_timeslot_bit_map = 2;
        cCfg.tsMap.line1_timeslot_bit_map = 0;
        cCfg.tsMap.line2_timeslot_bit_map = 0;
        cCfg.tsMap.line3_timeslot_bit_map = 0;
        cCfg.voiceIdleAction = 0;
        cCfg.voiceIdlePattern = 0x7E;
        cCfg.channelDataInvert =0;
        cCfg.channelBitEndianness = 1;
        cCfg.channelByteSwap = 0;
        config=ICP_HSSDRV_PORT_HMVIP_FRAMER_MEZZANINE_CONFIG;
        for(i=0;i<3;i++)
        {
                icp_FramerDrvInit(i);
                icp_FramerDrvConfigSet(i,ICP_FRAMERDRV_CFG_E1_CCS_HDB3_DATA);
                icp_FramerDrvLoopbackSet(i,ICP_FRAMERDRV_LOOPBACK_DIGITAL);
        }
}
void uninit()
{
        for(i=channelNum;i>0;i--)
        {
                ioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_DOWN,i);
        }
        for(j=channelNum;j>0;j--)
        {
                ioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_REMOVE,j);
        }
        for(i=0;i<3;i++)
        {
                if(portStatus[i] == UP)
                {
                        ioctl(fd[openCounter], ICP_HSSVOICEDRV_PORT_DOWN, 0);
                }
        }
        for(i=0;i<openCounter;i++)
        {
                close(fd[i]);
        }
        for(i=0;i<3;i++)
        {
                icp_FramerDrvUninit(i);
        }
}
/******DU testcases for portStatus******/
uint8_t test1(void)
{
        init();
        if(openCounter<128)
        {
                printf ("open: ResultExpected = 0\n");
                ResultExpected = 0;
                openCounter+=1;
        }
        else
        {
                printf ("open: Setting ResultExpected = -1\n");
                ResultExpected = -1;
        }
        fd[openCounter] = open("/dev/hss-voice",O_RDWR);
        if(fd[openCounter] < ResultExpected)
        {
                uninit();
                return FAIL;
        }

        cCfg.channelId = ++channelNum;
        cCfg.portId = portNum;
        cCfg.voicePacketSize = 240;
        cCfg.tsMap.line0_timeslot_bit_map = tsMap.line0_timeslot_bit_map;
        cCfg.tsMap.line1_timeslot_bit_map = tsMap.line1_timeslot_bit_map;
        cCfg.tsMap.line2_timeslot_bit_map = tsMap.line2_timeslot_bit_map;
        cCfg.tsMap.line3_timeslot_bit_map = tsMap.line3_timeslot_bit_map;
        cCfg.voiceIdleAction = 0;
        cCfg.voiceIdlePattern = 0x7E;
        cCfg.channelDataInvert =0;
        cCfg.channelBitEndianness = 1;
        cCfg.channelByteSwap = 0;
        if(channelsOnFd[openCounter]<128 &&
                portStatus[portNum]==UP &&
                (channelsUsed & channelNum) == 0 /*channel number is not used*/ &&
                (((tsUsed.line0_timeslot_bit_map & tsMap.line0_timeslot_bit_map) ==0)
&&   ((tsUsed.line1_timeslot_bit_map   &   tsMap.line1_timeslot_bit_map)   ==0)   &&
((tsUsed.line2_timeslot_bit_map     &     tsMap.line2_timeslot_bit_map)     ==0)     &&
((tsUsed.line3_timeslot_bit_map & tsMap.line3_timeslot_bit_map) ==0)) &&
                ((tsMap.line0_timeslot_bit_map        >=        0x00000002        &&
tsMap.line0_timeslot_bit_map <=0xFFFFFFFF) ||  /*ensure that the channel does not span
E1's*/(tsMap.line1_timeslot_bit_map  >=  0x00000002  &&  tsMap.line1_timeslot_bit_map
<=0xFFFFFFFF)     ||     (tsMap.line2_timeslot_bit_map     >=     0x00000002     &&
tsMap.line3_timeslot_bit_map   <=0xFFFFFFFF)   ||   (tsMap.line3_timeslot_bit_map   >=
0x00000002 && tsMap.line0_timeslot_bit_map <=0xFFFFFFFF)))
        {
                printf ("channelAdd: ResultExpected = 0\n");
                ResultExpected = 0;
                channelsOnFd[openCounter]++;
                channelConfig[channelNum].line0_timeslot_bit_map=
tsMap.line0_timeslot_bit_map;
                channelConfig[channelNum].line1_timeslot_bit_map=
tsMap.line1_timeslot_bit_map;
                channelConfig[channelNum].line2_timeslot_bit_map=
tsMap.line2_timeslot_bit_map;
                channelConfig[channelNum].line3_timeslot_bit_map=
tsMap.line3_timeslot_bit_map;
                tsUsed.line0_timeslot_bit_map |= tsMap.line0_timeslot_bit_map;
                tsUsed.line1_timeslot_bit_map |= tsMap.line1_timeslot_bit_map;
                tsUsed.line2_timeslot_bit_map |= tsMap.line2_timeslot_bit_map;
                tsUsed.line3_timeslot_bit_map |= tsMap.line3_timeslot_bit_map;
                channelsUsed |= channelNum;
                channelsOnPort[portNum]++;
        }
        else
        {
                printf ("channelAdd: Setting ResultExpected = -1\n");
                ResultExpected = -1;
        }
        tsMap.line0_timeslot_bit_map = tsMap.line0_timeslot_bit_map<<1;
        i= ioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_ADD, &cCfg);
        if(i!=0) channelNum--;
        if(i != ResultExpected)
        {
                printf ("Result: %d, was not expected\n", i);
                uninit();
                return FAIL;
        }
        uninit();
        return PASS;
}

uint8_t test2(void)
{
        init();
        if(openCounter<128)
        {
                printf ("open: ResultExpected = 0\n");
                ResultExpected = 0;
                openCounter+=1;
        }
        else
        {
                printf ("open: Setting ResultExpected = -1\n");
                ResultExpected = -1;
        }
        fd[openCounter] = open("/dev/hss-voice",O_RDWR);
        if(fd[openCounter] < ResultExpected)
        {
                uninit();
                return FAIL;
        }
        if((portConfig[portNum]==config || portConfig[portNum]==NOT_SET))
        {
                printf ("portUp: ResultExpected = 0\n");
                ResultExpected = 0;
                portConfig[portNum]=config;
                portStatus[portNum]=UP;
        }
        else
        {
                printf ("portUp: Setting ResultExpected = -1\n");
                ResultExpected = -1;
        }
        if(ioctl(fd[openCounter], ICP_HSSVOICEDRV_PORT_UP, &pCfg) != ResultExpected)
        {
                printf ("Result: %d, was not expected\n", i);
                uninit();
                return FAIL;
        }

        cCfg.channelId = ++channelNum;
        cCfg.portId = portNum;
        cCfg.voicePacketSize = 240;
        cCfg.tsMap.line0_timeslot_bit_map = tsMap.line0_timeslot_bit_map;
        cCfg.tsMap.line1_timeslot_bit_map = tsMap.line1_timeslot_bit_map;
        cCfg.tsMap.line2_timeslot_bit_map = tsMap.line2_timeslot_bit_map;
        cCfg.tsMap.line3_timeslot_bit_map = tsMap.line3_timeslot_bit_map;
        cCfg.voiceIdleAction = 0;
        cCfg.voiceIdlePattern = 0x7E;
        cCfg.channelDataInvert =0;
        cCfg.channelBitEndianness = 1;
        cCfg.channelByteSwap = 0;
        if(channelsOnFd[openCounter]<128 &&
                 portStatus[portNum]==UP &&
                 (channelsUsed & channelNum) == 0 /*channel number is not used*/ &&
                 (((tsUsed.line0_timeslot_bit_map & tsMap.line0_timeslot_bit_map) ==0)
&&   ((tsUsed.line1_timeslot_bit_map   &   tsMap.line1_timeslot_bit_map)   ==0)   &&
((tsUsed.line2_timeslot_bit_map     &     tsMap.line2_timeslot_bit_map)     ==0)     &&
((tsUsed.line3_timeslot_bit_map & tsMap.line3_timeslot_bit_map) ==0)) &&
                 ((tsMap.line0_timeslot_bit_map         >=         0x00000002         &&
tsMap.line0_timeslot_bit_map <=0xFFFFFFFF) ||  /*ensure that the channel does not span
E1's*/(tsMap.line1_timeslot_bit_map  >=  0x00000002  &&  tsMap.line1_timeslot_bit_map
<=0xFFFFFFFF)      ||      (tsMap.line2_timeslot_bit_map      >=      0x00000002      &&
tsMap.line3_timeslot_bit_map   <=0xFFFFFFFF)   ||   (tsMap.line3_timeslot_bit_map   >=
0x00000002 && tsMap.line0_timeslot_bit_map <=0xFFFFFFFF)))
        {
                printf ("channelAdd: ResultExpected = 0\n");
                ResultExpected = 0;
                channelsOnFd[openCounter]++;
                channelConfig[channelNum].line0_timeslot_bit_map=
tsMap.line0_timeslot_bit_map;
                channelConfig[channelNum].line1_timeslot_bit_map=
tsMap.line1_timeslot_bit_map;
                channelConfig[channelNum].line2_timeslot_bit_map=
tsMap.line2_timeslot_bit_map;
                channelConfig[channelNum].line3_timeslot_bit_map=
tsMap.line3_timeslot_bit_map;
                tsUsed.line0_timeslot_bit_map |= tsMap.line0_timeslot_bit_map;
                tsUsed.line1_timeslot_bit_map |= tsMap.line1_timeslot_bit_map;
                tsUsed.line2_timeslot_bit_map |= tsMap.line2_timeslot_bit_map;
                tsUsed.line3_timeslot_bit_map |= tsMap.line3_timeslot_bit_map;
                channelsUsed |= channelNum;
                channelsOnPort[portNum]++;
        }
        else
        {
                printf ("channelAdd: Setting ResultExpected = -1\n");
                ResultExpected = -1;
        }
        tsMap.line0_timeslot_bit_map = tsMap.line0_timeslot_bit_map<<1;
        i= ioctl(fd[openCounter], ICP_HSSVOICEDRV_CHAN_ADD, &cCfg);
        if(i!=0) channelNum--;
        if(i != ResultExpected)
        {
                printf ("Result: %d, was not expected\n", i);
                uninit();
                return FAIL;
        }
        uninit();
        return PASS;
}
```

## IX. Summary of Results from BVA Tests

Build 55 test Results

| test Case | Result | Comment |
|---|---|---|
| 1,4,6,7,8,10,11,13, 17,19,27 | Pass | |
| 2 | Fail | 128 clients with a single channel each: some failed due to repeated byte in loopback frame or<br>chunks 8 bytes of data missing in loopback frame<br>stuck in loop trying to remove channels at end of test |
| 3, 21, 23, 24,26 | Failed | 1 byte dropped out of rx frame |
| 22,25 | Fail | 1 byte repeated in of rx frame |
| 8 | Fail | section of 4 bytes missing in received framer |
| 12 | Fail | section of four bytes repeated in received frame |
| 14,16,18, 20 | Fail | section of 2 bytes repeated in received framer |
| 15. | Fail | section of 2 bytes missing in received framer |
| 28, 29 | Fail | Firmware Error bring up channels |

Version 1.0 test Results

| test Case | Result | Comment |
|---|---|---|
| 1-29 | Pass | |