Increasing Test Coverage on Current Intel Platforms

An Interactive and Automated Approach

by Larry Osborn

ASSET

Larry Osborn

Larry Osborn, Senior Product Manager, at ASSET InterTech, has over 30 years of experience in product management, hardware/software product design and development, product delivery to the marketplace and user support. Over the years, Larry has a proven track record of identifying user needs and opportunities in the marketplace, providing innovative solutions and exceeding the expectations of users. At ASSET, Larry is responsible for the profit and loss for a product group. Prior to ASSET, he has held positions with Lockheed Martin, OCD Systems, Wind River, Hewlett-Packard, Ford Aerospace and Intel® Corporation. He holds a Bachelor's Degree in Computer Science from the University of Kansas and various technical and marketing training certifications.

Table of Contents

Introduction

This eBook will discuss the use of functional testing in a production test environment to increase test coverage. First determining the definition of a functional test will need to be analyzed. Conducting a "google search" of functional testing results in a whole host of definitions. A good definition by Wikipedia below with the links removed.

> ***Functional testing*** *is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (not like in white-box testing). Functional testing usually describes what the system does.*
>
> *Functional testing does not imply that you are testing a function (method) of your module or class. Functional testing tests a slice of the functionality of the whole system.*
>
> *Functional testing differs from system testing in that functional testing "verifies a program by checking it against ... design document(s) or specification(s)", while system testing "validate[s] a program by checking it against the published user or system requirements" (Kaner, Falk, Nguyen 1999, p. 52).*
>
> *Functional testing has many types:*
> - *Smoke testing*
> - *Sanity testing*
> - *Regression testing*
> - *Usability testing*

Most of the definitions found on a google search are about how to test software functions (e.g. test how the software behaves or functions). In the context of this eBook, the use of functional testing does not adhere to that definition. Here, a functional test is used as a means to test hardware, at-speed, via software running on the processor.

By developing a functional test on a golden platform (known working hardware), it stands to reason that the same software can be used to test an identical platform in production to identify functional hardware failures.

Some manufacturers are using this technique by executing the BIOS and then preforming a functional test, thus using the BIOS as a functional test setup, although it might not be described or considered as such. Of course, the main purpose of the BIOS is to boot, load, and initialize all the capabilities of the board, thus the purpose is not to be a high-quality functional test. The assumptions about the BIOS ability to "test" hardware is overreaching for the requirements of the BIOS. A good discussion on this topic is in the previous eBook in this series called "*Testing Memories without BIOS on the Latest Intel® Platforms*". The focus of this eBook will show how functional testing can be applied to increase test coverage of the unit under test (UUT) leveraging the processor as the smartest embedded instrument on the board. This technique should be as part of an overall test methodology that also includes a structural test. The reason for combining the two distinct methods, functional and structural, is to provide maximum test coverage of a given platform. In the context of this eBook, we are talking about Intel platforms.

Interactive Approach: Functional Test Development Methods and Steps

Before starting the test development process, it is necessary to analyze the test objectives, steps, and tools necessary to meet the objectives. To have maximum control over the test process, while conducting a functional test, the CPU will act as the control agent; therefore the tool selection is important. The tool(s) needs to allow the test to start and stop the processor, run test code or scripts, allow for examination of memory, registers, pin signals and IO space, and provide for user input/output in the form of logs or graphical user interface.

The tool used to meet the needs mentioned above in the development of this eBook is ASSET's ScanWorks® Processor-Controlled Test (PCT). Although this eBook provides a manual methodical approach of test development, it should be noted that the tool used here can automate this process by simply connecting to a known working board and run the PCT Golden Board Interrogator utility and then run the Automated Test Generation utility.

As with all engineering tasks, breaking the test development process into manageable steps is the best approach. To construct a programmatic test implementation, it is often necessary to start with an interactive approach to test development. This interactive approach allows the developer, when connected to a known working board with a superior tool, to conduct experiments on the target platform to sharpen the skills needed for programmatic development. The recommended approach consists of three top-level steps:

- Understanding the Board Structure
- Fault Isolation
- Fault Identification

Understanding the Board Structure

Using available platform documentation, divide the board into functional blocks and draw a block diagram of the board structure similar to the one shown in Figure 1.

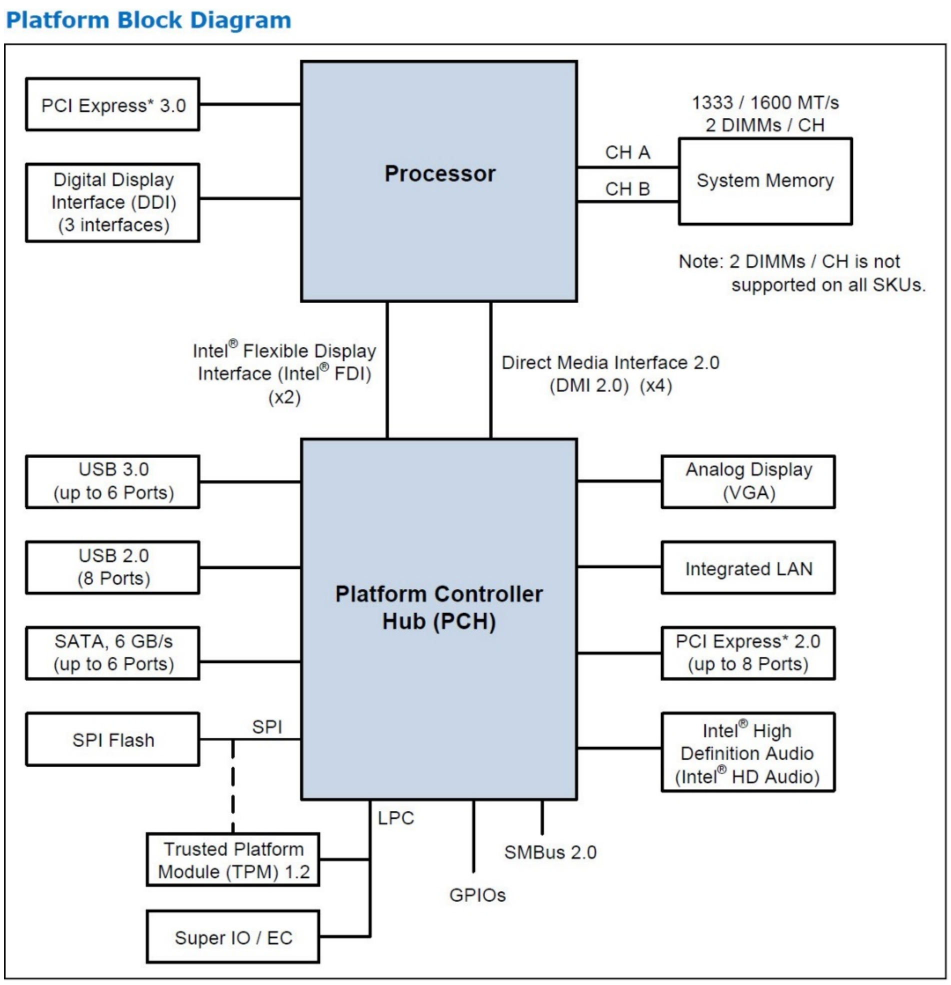

Platform Block Diagram
PCI Express* 3.0
Digital Display Interface (DDI) (3 interfaces)
Processor
1333 / 1600 MT/s 2 DIMMs / CH
CH A
CH B
System Memory
Note: 2 DIMMs / CH is not supported on all SKUs.
Intel® Flexible Display Interface (Intel® FDI) (x2)
Direct Media Interface 2.0 (DMI 2.0) (x4)
USB 3.0 (up to 6 Ports)
USB 2.0 (8 Ports)
SATA, 6 GB/s (up to 6 Ports)
SPI Flash
SPI
Platform Controller Hub (PCH)
Analog Display (VGA)
Integrated LAN
PCI Express* 2.0 (up to 8 Ports)
Intel® High Definition Audio (Intel® HD Audio)
LPC
SMBus 2.0
GPIOs
Trusted Platform Module (TPM) 1.2
Super IO / EC

Figure 1: Example Platform Architecture

1. Identify the platforms main processor and PCH.
2. Identify major buses and their relationships (DMI, PCIe SPI, LAN, SATA, USB, etc.).
3. Identify board memory and assign addresses.
4. Identify major peripheral devices, I/O components which drive them, and the buses which connect to the I/O devices. Define what range(s) an IO controller will use for ALL its IO functions, and defined the address ranges, then for each IO function in the block, define the I/O addresses.

From the block diagram in Figure 1, the functional blocks can be identified. The block diagram is not a complete representation of the platform, but provides enough information about the platform for test development. In the current Intel designs, most of the functional blocks are connected to the CPU complex, consisting of the CPU and PCH (Processor Control Hub). There are three functional blocks that need testing on the processor, which are the PCIe 3.0, memory interfaces (both channels), and the DMI interface. The rest of the testing is conducted via the PCH. To be clear, in the communications about the role of the processor and the PCH, the testing is controlled by the processor, but the configuration of the PCH is what enables access to the downstream components. Beyond the PCH, the developer will need to collect available datasheets for all the major board ICs. The best source for these is the websites of the IC manufacturers.

Fault Isolation

Once the functional blocks are identified, testing should commence from the processor down to verify each functional block; a test sequence can be developed using PCT's pre-programmed functions. Functional testing implies that the device is acted upon in such a way to produce a response from a given stimuli. For bus testing, reading and writing a register or memory location with known expected response will suffice for a bus test. For I/O devices it might be sufficient to read a status register of the I/O component to know that the board device structure and components are indeed functional. It should be noted that a more exhaustive test maybe required, but that decision is left up to the test developer.

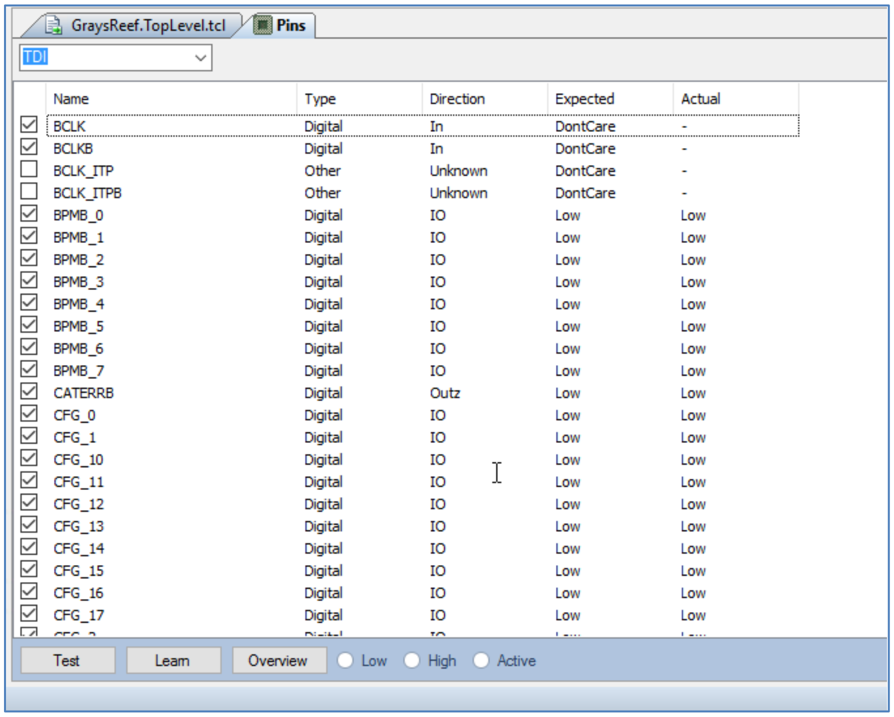
For this discussion, functional blocks and the test sequences used can be categorized as follows:

- Processor Area
- Memory
- PCIe
- PCH

Processor Area

Working with a known working board, the developer needs to understand when a platform under test has a problem with the debug communication. If the processor can reset and enter test mode (i.e. debug port communications mode can be established), it is safe to assume that the processor area is OK. However, the developer needs to assume that there could be a problem with the debug communications, otherwise, on a failing board the functional programmatic test could never begin. To address this concern, the developer needs to include a structural test of the processor. PCT provides the means of conducting a structural test of the processor.

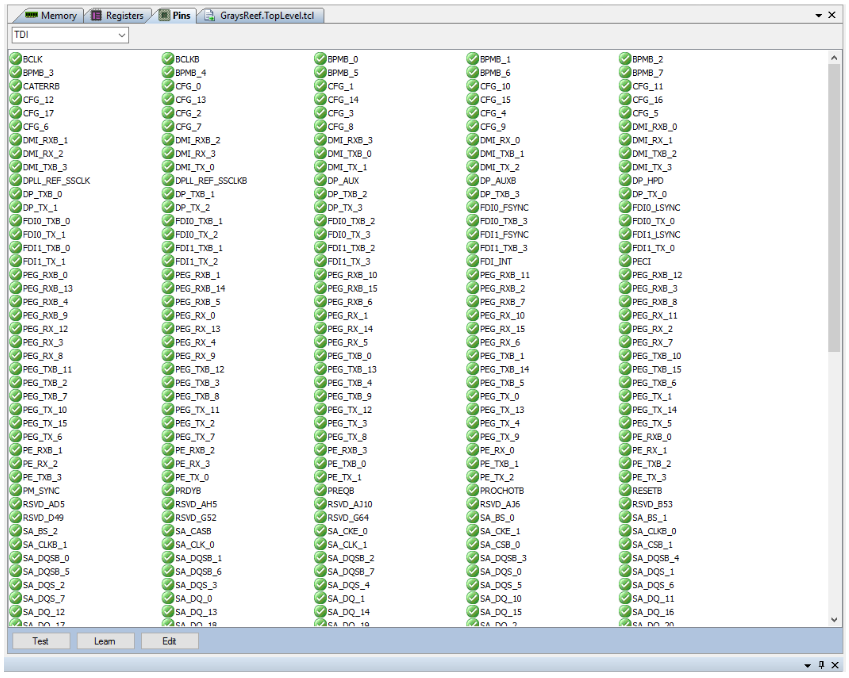
A sample needs to be taken when the processor pins are in a known state and the signals are quiescent from a known working system. The best place to learn the signal state is the reset state. Figure 2 is an example of sampling the processor pin status at reset from the PCT pins view. Not all the processor pins are selected or shown and this interface allows the developer to decide which pins need to be sampled.



	Name	Type	Direction	Expected	Actual
✓	BCLK	Digital	In	DontCare	-
✓	BCLKB	Digital	In	DontCare	-
	BCLK_ITP	Other	Unknown	DontCare	-
	BCLK_ITPB	Other	Unknown	DontCare	-
✓	BPMB_0	Digital	IO	Low	Low
✓	BPMB_1	Digital	IO	Low	Low
✓	BPMB_2	Digital	IO	Low	Low
✓	BPMB_3	Digital	IO	Low	Low
✓	BPMB_4	Digital	IO	Low	Low
✓	BPMB_5	Digital	IO	Low	Low
✓	BPMB_6	Digital	IO	Low	Low
✓	BPMB_7	Digital	IO	Low	Low
✓	CATERRB	Digital	Outz	Low	Low
✓	CFG_0	Digital	IO	Low	Low
✓	CFG_1	Digital	IO	Low	Low
✓	CFG_10	Digital	IO	Low	Low
✓	CFG_11	Digital	IO	Low	Low
✓	CFG_12	Digital	IO	Low	Low
✓	CFG_13	Digital	IO	Low	Low
✓	CFG_14	Digital	IO	Low	Low
✓	CFG_15	Digital	IO	Low	Low
✓	CFG_16	Digital	IO	Low	Low
✓	CFG_17	Digital	IO	Low	Low

Figure 2: Pins Edit View

By sampling the processor pins state at reset of a golden target platform, the platform under test can be compared to the expected state. Figure 3 is a comparison overview, where the system is verified to be in the correct state, and the pin connectivity is correct.



✓ BCLK	✓ BCLKB	✓ BPMB_0	✓ BPMB_1	✓ BPMB_2
✓ BPMB_3	✓ BPMB_4	✓ BPMB_5	✓ BPMB_6	✓ BPMB_7
✓ CATERRB	✓ CFG_0	✓ CFG_1	✓ CFG_10	✓ CFG_11
✓ CFG_12	✓ CFG_13	✓ CFG_14	✓ CFG_15	✓ CFG_16
✓ CFG_17	✓ CFG_2	✓ CFG_3	✓ CFG_4	✓ CFG_5
✓ CFG_6	✓ CFG_7	✓ CFG_8	✓ CFG_9	✓ DMI_RXB_0
✓ DMI_RXB_1	✓ DMI_RXB_2	✓ DMI_RXB_3	✓ DMI_RX_0	✓ DMI_RX_1
✓ DMI_RX_2	✓ DMI_RX_3	✓ DMI_TXB_0	✓ DMI_TXB_1	✓ DMI_TXB_2
✓ DMI_TXB_3	✓ DMI_TX_0	✓ DMI_TX_1	✓ DMI_TX_2	✓ DMI_TX_3
✓ DPLL_REF_SSCLK	✓ DPLL_REF_SSCLKB	✓ DP_AUX	✓ DP_AUXB	✓ DP_HPD
✓ DP_TXB_0	✓ DP_TXB_1	✓ DP_TXB_2	✓ DP_TXB_3	✓ DP_TX_0
✓ DP_TX_1	✓ DP_TX_2	✓ DP_TX_3	✓ FDI0_FSYNC	✓ FDI0_LSYNC
✓ FDI0_TXB_0	✓ FDI0_TXB_1	✓ FDI0_TXB_2	✓ FDI0_TXB_3	✓ FDI0_TX_0
✓ FDI0_TX_1	✓ FDI0_TX_2	✓ FDI0_TX_3	✓ FDI1_FSYNC	✓ FDI1_LSYNC
✓ FDI1_TXB_0	✓ FDI1_TXB_1	✓ FDI1_TXB_2	✓ FDI1_TXB_3	✓ FDI1_TX_0
✓ FDI1_TX_1	✓ FDI1_TX_2	✓ FDI1_TX_3	✓ FDI_INT	✓ PECI
✓ PEG_RXB_0	✓ PEG_RXB_1	✓ PEG_RXB_10	✓ PEG_RXB_11	✓ PEG_RXB_12
✓ PEG_RXB_13	✓ PEG_RXB_14	✓ PEG_RXB_15	✓ PEG_RXB_2	✓ PEG_RXB_3
✓ PEG_RXB_4	✓ PEG_RXB_5	✓ PEG_RXB_6	✓ PEG_RXB_7	✓ PEG_RXB_8
✓ PEG_RXB_9	✓ PEG_RX_0	✓ PEG_RX_1	✓ PEG_RX_10	✓ PEG_RX_11
✓ PEG_RX_12	✓ PEG_RX_13	✓ PEG_RX_14	✓ PEG_RX_15	✓ PEG_RX_2
✓ PEG_RX_3	✓ PEG_RX_4	✓ PEG_RX_5	✓ PEG_RX_6	✓ PEG_RX_7
✓ PEG_RX_8	✓ PEG_RX_9	✓ PEG_TXB_0	✓ PEG_TXB_1	✓ PEG_TXB_10
✓ PEG_TXB_11	✓ PEG_TXB_12	✓ PEG_TXB_13	✓ PEG_TXB_14	✓ PEG_TXB_15
✓ PEG_TXB_2	✓ PEG_TXB_3	✓ PEG_TXB_4	✓ PEG_TXB_5	✓ PEG_TXB_6
✓ PEG_TXB_7	✓ PEG_TXB_8	✓ PEG_TXB_9	✓ PEG_TX_0	✓ PEG_TX_1
✓ PEG_TX_10	✓ PEG_TX_11	✓ PEG_TX_12	✓ PEG_TX_13	✓ PEG_TX_14
✓ PEG_TX_15	✓ PEG_TX_2	✓ PEG_TX_3	✓ PEG_TX_4	✓ PEG_TX_5
✓ PEG_TX_6	✓ PEG_TX_7	✓ PEG_TX_8	✓ PEG_TX_9	✓ PE_RXB_0
✓ PE_RXB_1	✓ PE_RXB_2	✓ PE_RXB_3	✓ PE_RX_0	✓ PE_RX_1
✓ PE_RX_2	✓ PE_RX_3	✓ PE_TXB_0	✓ PE_TXB_1	✓ PE_TXB_2
✓ PE_TXB_3	✓ PE_TX_0	✓ PE_TX_1	✓ PE_TX_2	✓ PE_TX_3
✓ PM_SYNC	✓ PRDYB	✓ PREQB	✓ PROCHOTB	✓ RESETB
✓ RSVD_AD5	✓ RSVD_AH5	✓ RSVD_AJ10	✓ RSVD_AJ6	✓ RSVD_B53
✓ RSVD_D49	✓ RSVD_G52	✓ RSVD_G64	✓ SA_BS_0	✓ SA_BS_1
✓ SA_BS_2	✓ SA_CASB	✓ SA_CKE_0	✓ SA_CKE_1	✓ SA_CLKB_0
✓ SA_CLKB_1	✓ SA_CLK_0	✓ SA_CLK_1	✓ SA_CSB_0	✓ SA_CSB_1
✓ SA_DQSB_0	✓ SA_DQSB_1	✓ SA_DQSB_2	✓ SA_DQSB_3	✓ SA_DQSB_4
✓ SA_DQSB_5	✓ SA_DQSB_6	✓ SA_DQSB_7	✓ SA_DQS_0	✓ SA_DQS_1
✓ SA_DQS_2	✓ SA_DQS_3	✓ SA_DQS_4	✓ SA_DQS_5	✓ SA_DQS_6
✓ SA_DQS_7	✓ SA_DQ_0	✓ SA_DQ_1	✓ SA_DQ_10	✓ SA_DQ_11
✓ SA_DQ_12	✓ SA_DQ_13	✓ SA_DQ_14	✓ SA_DQ_15	✓ SA_DQ_16

Figure 3: Pins Test Results

Using the data from this pins test, the developer can include the expected pins status results with the test profile. Other buses like DMI need to be considered as a candidate for structural sampling on a golden system and use the collected data to verify the DMI interface between the processor and the PCH. Pin testing on the PCH can be accomplished provided that the PCH and processor are on the same JTAG scan chain. The objective is to ensure the structural integrity between these two devices before proceeding to the functional check out of the platform. The use of pins testing, when the state of the device in question is known, is a good technique when a functional test fails or amplification of the data is needed.

The next area of the processor block that needs testing is the memory block.

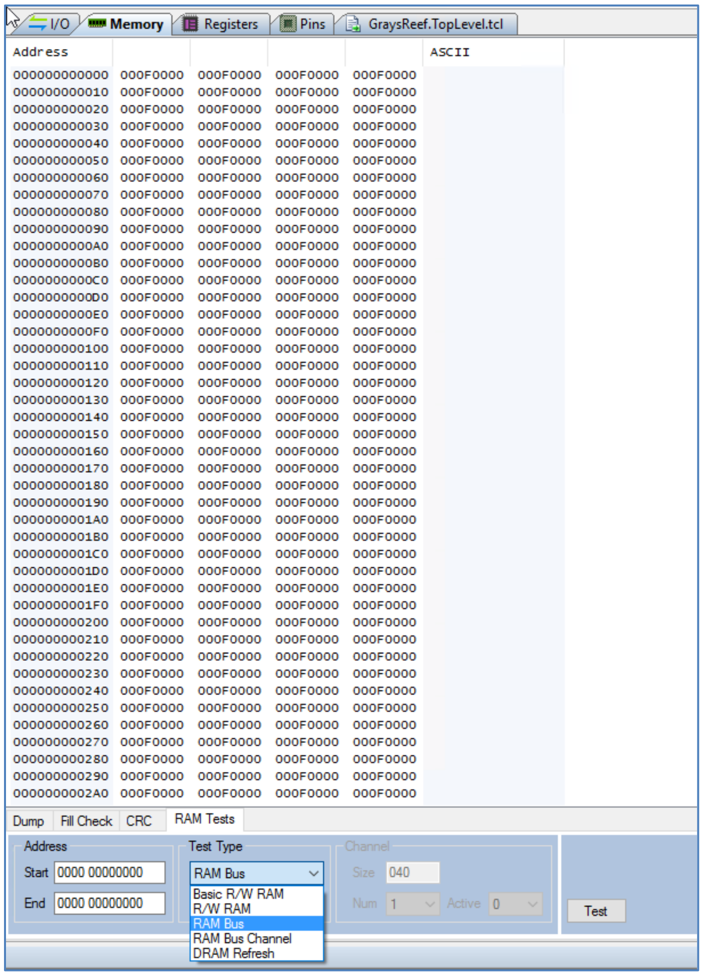
Memory

Current Intel platform architectures, like the one depicted here, requires that the memory controller is configured or setup first before memory testing can begin. There are two ways to do this. Executing the platform BIOS, found in the SPI flash, or by writing the configuration registers within the memory controller directly.

Starting with BIOS execution. The SPI flash contains several firmware elements including the BIOS. Within the BIOS is the memory setup code. This code, referred to as Memory Reference Code (MRC), does the memory controller setup and lane training such that the memory is in a functional state whereby the testing can begin. However, from a functional test development point-of-view, the SPI interface from the PCH has yet to be tested. The developer should consider what would happen, in the production line, if the SPI flash was corrupted or the SPI flash was replaced with a different device type that the flashing process didn't handle this device properly. The structural aspects of the SPI interface must be considered, if possible, before assuming that the functional test can execute.

In the case of memories, supporting a structural test is not always possible. Memory devices often lack the boundary scan cell necessary to support structural testing. So with no validated SPI access and no structural means of testing the memory, the alternate is writing (directly) to the memory controller registers. As pointed out earlier, the MRC is the code that does this configuration, and that code is a closely held secret by Intel due to the proprietary nature of the memory setup. Directly writing to the memory control registers is not an option due to the MRC constraints. Is there a course of action that the developer can take to complete the testing of the processor block? Namely the memory block.

PCT provides a means of using the CPU's internal cache as a RAM so MRC code can be executed from the cache. Developed by ASSET and built into the PCT product is an instrumented version of the MRC, which is executed out of cache and used to conduct the memory setup. After using this setup, the developer can write a memory test. Developers can use an interactive means of examining (testing) memory within PCT. Shown in Figure 4 are a memory dump and the functional tests that can be executed against this memory region once the memory setup has been completed.



```
Address                                                 ASCII
000000000000  000F0000  000F0000  000F0000  000F0000
000000000010  000F0000  000F0000  000F0000  000F0000
000000000020  000F0000  000F0000  000F0000  000F0000
000000000030  000F0000  000F0000  000F0000  000F0000
000000000040  000F0000  000F0000  000F0000  000F0000
000000000050  000F0000  000F0000  000F0000  000F0000
000000000060  000F0000  000F0000  000F0000  000F0000
000000000070  000F0000  000F0000  000F0000  000F0000
000000000080  000F0000  000F0000  000F0000  000F0000
000000000090  000F0000  000F0000  000F0000  000F0000
0000000000A0  000F0000  000F0000  000F0000  000F0000
0000000000B0  000F0000  000F0000  000F0000  000F0000
0000000000C0  000F0000  000F0000  000F0000  000F0000
0000000000D0  000F0000  000F0000  000F0000  000F0000
0000000000E0  000F0000  000F0000  000F0000  000F0000
0000000000F0  000F0000  000F0000  000F0000  000F0000
000000000100  000F0000  000F0000  000F0000  000F0000
000000000110  000F0000  000F0000  000F0000  000F0000
000000000120  000F0000  000F0000  000F0000  000F0000
000000000130  000F0000  000F0000  000F0000  000F0000
000000000140  000F0000  000F0000  000F0000  000F0000
000000000150  000F0000  000F0000  000F0000  000F0000
000000000160  000F0000  000F0000  000F0000  000F0000
000000000170  000F0000  000F0000  000F0000  000F0000
000000000180  000F0000  000F0000  000F0000  000F0000
000000000190  000F0000  000F0000  000F0000  000F0000
0000000001A0  000F0000  000F0000  000F0000  000F0000
0000000001B0  000F0000  000F0000  000F0000  000F0000
0000000001C0  000F0000  000F0000  000F0000  000F0000
0000000001D0  000F0000  000F0000  000F0000  000F0000
0000000001E0  000F0000  000F0000  000F0000  000F0000
0000000001F0  000F0000  000F0000  000F0000  000F0000
000000000200  000F0000  000F0000  000F0000  000F0000
000000000210  000F0000  000F0000  000F0000  000F0000
000000000220  000F0000  000F0000  000F0000  000F0000
000000000230  000F0000  000F0000  000F0000  000F0000
000000000240  000F0000  000F0000  000F0000  000F0000
000000000250  000F0000  000F0000  000F0000  000F0000
000000000260  000F0000  000F0000  000F0000  000F0000
000000000270  000F0000  000F0000  000F0000  000F0000
000000000280  000F0000  000F0000  000F0000  000F0000
000000000290  000F0000  000F0000  000F0000  000F0000
0000000002A0  000F0000  000F0000  000F0000  000F0000
```

Address
Start: 0000 00000000
End: 0000 00000000

Test Type: RAM Bus
- Basic R/W RAM
- R/W RAM
- RAM Bus
- RAM Bus Channel
- DRAM Refresh

Channel
Size: 040
Num: 1 Active: 0

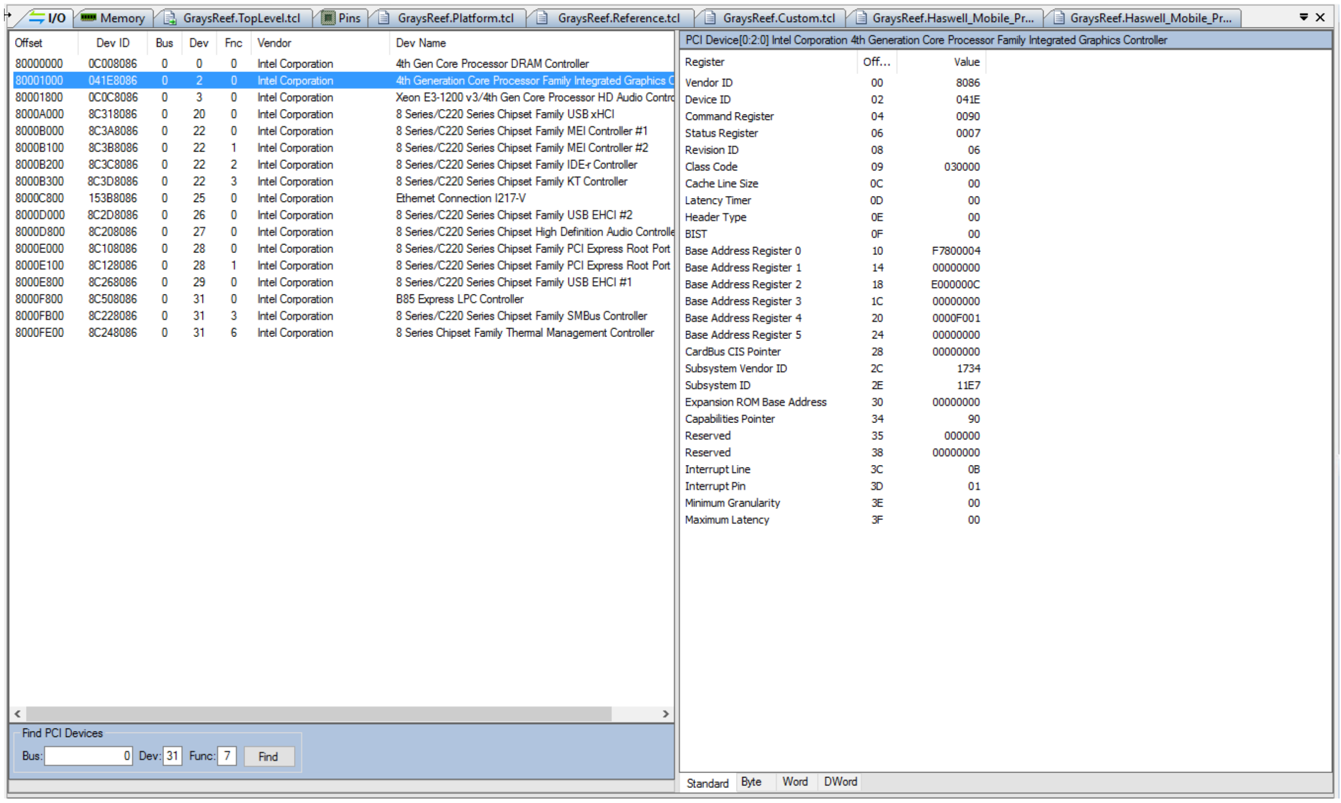
Figure 4: Interactive Memory Test

This interactive testing is a good way of validation of the parameters needed for the programmatic test like the number of channels, the number of DIMMS per channel, the size of the DIMMS and others. The developer can ensure that all areas of memory are accounted for in the development process. The data then needs to be extracted and used in the memory test sequence. Often there are several memory population schemes that need to be considered for a given platform. Using this development process, the developer can vary the memory population densities and have tests that are configuration specific. Then using data about the memory population during a read of a configuration register determine which functional test should be performed. Once all the memory parameters have been identified, PCT reports the failures to pinpoint the channel, rank, DIMM, device and not just the failing address.

The final functional block of the CPU is the PCIe component.

PCIe

Here the developer needs to know what PCIe devices are populated. PCI device functions have a configuration space which is addressable by knowing the 8-bit bus, 3-bit device and 5-bit function numbers for the device (commonly referred to as the BDF bus/device/function). This BDF data allows up to 256 buses, each with up to 32 devices, each supporting eight functions. Still using the golden board as a development platform and armed with the information about what PCI devices are populated, the developer can conduct experiments to understand what information is available to verify the correct device is connected to the bus and verify, to the extent possible, its operation. Figure 5 is an example of a PCI dump of bus 0, devices 0-31, functions 0-7.



Offset	Dev ID	Bus	Dev	Fnc	Vendor	Dev Name
80000000	0C008086	0	0	0	Intel Corporation	4th Gen Core Processor DRAM Controller
80001000	041E8086	0	2	0	Intel Corporation	4th Generation Core Processor Family Integrated Graphics C
80001800	0C0C8086	0	3	0	Intel Corporation	Xeon E3-1200 v3/4th Gen Core Processor HD Audio Contro
8000A000	8C318086	0	20	0	Intel Corporation	8 Series/C220 Series Chipset Family USB xHCI
8000B000	8C3A8086	0	22	0	Intel Corporation	8 Series/C220 Series Chipset Family MEI Controller #1
8000B100	8C3B8086	0	22	1	Intel Corporation	8 Series/C220 Series Chipset Family MEI Controller #2
8000B200	8C3C8086	0	22	2	Intel Corporation	8 Series/C220 Series Chipset Family IDE-r Controller
8000B300	8C3D8086	0	22	3	Intel Corporation	8 Series/C220 Series Chipset Family KT Controller
8000C800	153B8086	0	25	0	Intel Corporation	Ethernet Connection I217-V
8000D000	8C2D8086	0	26	0	Intel Corporation	8 Series/C220 Series Chipset Family USB EHCI #2
8000D800	8C208086	0	27	0	Intel Corporation	8 Series/C220 Series Chipset High Definition Audio Controlle
8000E000	8C108086	0	28	0	Intel Corporation	8 Series/C220 Series Chipset Family PCI Express Root Port
8000E100	8C128086	0	28	1	Intel Corporation	8 Series/C220 Series Chipset Family PCI Express Root Port
8000E800	8C268086	0	29	0	Intel Corporation	8 Series/C220 Series Chipset Family USB EHCI #1
8000F800	8C508086	0	31	0	Intel Corporation	B85 Express LPC Controller
8000FB00	8C228086	0	31	3	Intel Corporation	8 Series/C220 Series Chipset Family SMBus Controller
8000FE00	8C248086	0	31	6	Intel Corporation	8 Series Chipset Family Thermal Management Controller

PCI Device[0:2:0] Intel Corporation 4th Generation Core Processor Family Integrated Graphics Controller

Register	Off...	Value
Vendor ID	00	8086
Device ID	02	041E
Command Register	04	0090
Status Register	06	0007
Revision ID	08	06
Class Code	09	030000
Cache Line Size	0C	00
Latency Timer	0D	00
Header Type	0E	00
BIST	0F	00
Base Address Register 0	10	F7800004
Base Address Register 1	14	00000000
Base Address Register 2	18	E000000C
Base Address Register 3	1C	00000000
Base Address Register 4	20	0000F001
Base Address Register 5	24	00000000
CardBus CIS Pointer	28	00000000
Subsystem Vendor ID	2C	1734
Subsystem ID	2E	11E7
Expansion ROM Base Address	30	00000000
Capabilities Pointer	34	90
Reserved	35	000000
Reserved	38	00000000
Interrupt Line	3C	0B
Interrupt Pin	3D	01
Minimum Granularity	3E	00
Maximum Latency	3F	00

Figure 5: PCI Bus 0

From the data collected, the developer can verify that the vendor ID and device ID matches what was documented in the device or platform manual. This data can then be use to develop the programmatic script. The status register can also be sampled. Comparing these known good values to the platform under test will provide the information about the correct part and functionality of the device. All the PCI devices need to be enumerated to be included in the test. This test development method can be a very time-consuming effort. There is a better more efficient way of dealing with the generating volumes of code manually. That better approach will be discussed at the end of this eBook.

PCH

The PCH provides access to several buses, SPI, SMBus, LPC, GPIO, SATA, USB, Integrated LAN, and PCIe. This eBook will not cover all buses or devices, but will instead focus on a couple of critical buses to help the developer understand what action needs to be taken. For the buses not covered, the information provided to this point in the eBook should provide enough information for the developer to develop a robust, structured functional test. As discussed earlier in the processor section, the DMI interface needs to be considered a candidate for structural sampling on a golden system and use the collected data to verify the DMI interface between the processor and the PCH. Pin testing on the PCH can be accomplished provided that the PCH and Processor are on the same JTAG scan chain. At this point, it is imperative the DMI interface is working. It is good practice to conduct a pins test on the PCH.

SPI bus.

The most critical bus on the PCH is the SPI bus. Without a functioning SPI interface, the board won't power up. Here is the catch-22 on the SPI bus and the DMI bus for that matter. For the processor to get to the point where the functional test can be conducted, the PCH needs to read the straps on the GPIO interface and other pin straps in addition execute the ME code within the SPI. Without the execution of the ME, the platform won't power up. So if either the SPI or the DMI link is broken, there is no functional test capability.

When validating the SPI bus, it is possible to conduct a pins test of the serial interface. Once the structural interface has been established, the developer should validate the contents of the SPI flash. The most expedient way to validate the contents of the flash is to perform a CRC test on the SPI bus with the address range of the BIOS and compare to a known good CRC.

SMBus

Another bus that deserves a structural test is that of the SMBus. The SMBus controls many devices on Intel platforms. One critical piece of data needed for testing is the SPD data which is only accessible via the SMBus. The SPD data from the EEPOM of the DIMM from the golden board should be extracted and used in comparison if and only if the platforms under test are using the same DIMMS. The data stored in the SPD is critical for memory setup and testing.

Other Buses

The developer will need to determine what buses have critical components attached and from that analysis determine if a structural test is necessary before beginning the functional test development.

Fault Identification

During test development, a test profile is created and a checkout of the first functional test profile should be conducted against a platform with known problems. This step is necessary to ensure that the functional test is robust enough to capture these known faults. During that exercise, the defective functional blocks isolated using the methods described above, will probably still contain a few components and numerous interconnects. To identify the actual defect within the failed functional block use the following methods.

- Visual Inspection – Keep it simple! Perform a detailed visual inspection of the isolated area. Look for shorts, bad solder joints, "tombstoned" resistors, wrong, ICs, etc.
- Tactile Inspection – Again, keep it simple! Are there any overheating ICs? Loop the failing test. Press down on all major ICs. Does the test pass now? This method can detect open circuits on devices such as BGAs etc.
- Probing – PCT in the interactive views can be used to create stimuli within a device so that probing via an oscilloscope can be used to determine a fault or signal quality.

Automated Approach: Functional Test Development Methods - A Better Way

From reading this eBook, the developer can be overwhelmed with the effort involved to create a robust functional test. Throughout the eBook, there are hints about a potentially better, time-saving, development approach. All of the development efforts start with a golden board/platform so that experiments can be executed against this platform to find what the correct register data values or setting a device state for the production tests. Then extract and use that date to create a programmatic (script) for the CPU to execute. PCT provides a better approach than the manual methodical development methods describe here.

Again starting with a golden board, boot the board to the EFI shell or Windows®. At this state all the devices are properly configured, then extract all the register settings and use that data as input to create a test. PCT has a Golden Board Interrogation (GBI) utility that, via JTAG run-control, does this extraction. PCT is model driven and from the data extracted in the GBI execution that data can be compared to existing platform models. The model combined with the GBI extracted data is used by the Automated Test Generation (ATG) Utility to create the Tcl profile.

A developer might recognize a problem with this approach when it comes to dealing with the MRC. The MRC is an algorithm that does repetitive reads/writes to the memory control registers to get the memory lane training to operate at maximum efficiency. Reading the last known state is not going to provide the optimal memory response for a platform under test. In electrical validation circles, this is referred to as the eye. Which is correct. The platform under test is not required to operate at the optimum memory access bandwidth for testing. The platform requirement is that the memory controller is configured so that a memory test, like RAMBUSTEST, can execute.

The ATG creates all the Tcl script needed to test the platform. These scripts can then be edited by the developer and adjusted as needed. In dealing with the PCIe enumeration, this will save the developer potentially days (weeks) of work.

Once the functional profile is generated from the ATG, the developer should determine what areas of the platform such as JTAG, DMI, SPI and others, would benefit from a structural and include pins test for these buses. The reason PCT doesn’t do this via the GBI/ATG is the pin state of a given devices is platform state specific. Pin state information within the model wouldn’t be relative to the pin state of the platform under test.

Summary

A tool designed for functional board test is invaluable in saving development effort and time. By following a simple 3-step procedure, defects can be rapidly isolated. By skipping to step 4, even more, savings can be realized.

1. Understand board structure. Break it into functional blocks and create a troubleshooting tree.
2. Isolate the fault area by using simple test sequences to verify each functional block.
3. Finally, identify the actual defect using a combination of inspection and probing.
4. Use the easiest development approach.

Learn More

Learn more about Processor-Controlled Test products on our website.