SCANSTA101 STA Master

Design Guide

2010 Revision 1.0

Developing a System with Embedded IEEE 1149.1 Boundary-Scan Self-Test

national.com/scan

National Semiconductor

Table of Contents

Acknowledgements **4**
A Word about the Automatic Test Pattern Generator Tools **5**
1. Introduction and Scope **6**
1.1. About this Design Guide 6
1.2. An Engineer's Historical Perspective 6
1.3. Purpose of this Design Guide 7
2. A Brief Introduction to Boundary Scan **8**
2.1. National's Boundary Scan Family 8
2.2. The IEEE 1149.1 Standard 8
2.3. Boundary Scan Basics 9
2.3.1. Elements of the IEEE 1149.1 Test Access Port (TAP) 9
2.3.2. Architecture of the IEEE 1149.1 TAP 9
2.3.3. Use of the IEEE 1149.1 TAP 12
2.4. Summary and Conclusions 13
3. Test System Description **14**
3.1. The SCANSTAEVK Demonstration Kit 14
3.1.1. PCI-1149.1/101 SCANSTA101 PC Card 14
3.1.2. SCANSTAEVK Demonstration Kit Backplane 14
3.1.3. SCANSTA111 Intermediate Board 15
3.1.4. Target SerDes Board 17
3.1.5. Demonstration Kit Summary 19
3.2. Test Development and Deployment Software 19
3.2.1. Automatic Test Pattern Generation 19
3.2.2. Conversion to EVF2 Format 20
3.2.3. Embedded Platform Software 20
3.2.4. ScanVec 21
3.3. Summary 21
4. Hardware Design Considerations for Built-In Self-Test **22**
4.1. System Function and Built-In Self-Test 22
4.1.1. Selection of JTAG-Enabled Devices 22
4.1.2. Selection of JTAG Support Devices 22
4.1.3. Delivery of the Test Vectors 23
4.1.4. TAP Connections 23
4.1.4.1. Single JTAG TAP over the Backplane 23
4.1.4.2. Multiple JTAG TAPs over the Backplane 24
4.1.4.3. Parallel-Port Communication over the Backplane 25
4.2. Test Implementation 27
4.2.1. Place and Route the JTAG Support Devices 27
4.2.2. Connect the Parallel Processor Interface of the SCANSTA101 STA Master(s) 27
4.2.3. Connect the TAP 28
4.2.4. Asynchronous Reset TRST* 28
4.2.5. Test Clock (TCK) 28

Table of Contents

4.2.6. Test Mode Select (TMS) ... 28
4.2.7. Test Data In (TDI) and Test Data Out (TDO) ... 29
4.3. Summary ... 30
5. Development of a Boundary Scan Test Pattern ... 31
5.1. ATPG Tool Selection ... 31
5.2. Producing the Netlist ... 31
5.3. Entering the Netlist into the ATPG Tool ... 32
5.4. Adding Information to the Netlist ... 33
5.4.1. Power and Ground Nets ... 33
5.4.2. Resistors ... 33
5.4.3. Transparent Devices ... 34
5.4.4. Adding BSDL Models ... 34
5.4.5. Modifying the Connections in the Netlist ... 36
5.5. Generation of the Boundary Scan Test and SVF File ... 37
5.5.1. Boundary Scan Test Strategy ... 37
5.5.2. Serial Vector Format ... 37
5.6. Generation of an EVF2 File using EVF Workbench ... 39
6. Embedded Vector Delivery Software ... 41
6.1. Development of the System Controller Software ... 41
6.1.1. National Semiconductor Code ... 41
6.1.2. Interface to the EVF2 File and Error Handling ... 42
6.1.2.1. EVF2 File Read Function ... 42
6.1.2.2. Error Handler ... 43
6.1.2.3. Fail Handler ... 43
6.1.2.4. Register Trace Handler ... 43
6.1.2.5. Debug Handler ... 43
6.1.2.6. Polling Handler ... 43
6.1.2.7. Timeout Value ... 44
6.1.2.8. User Pointer Arguments ... 44
6.2. Interface to the SCANSTA101 Master Registers ... 45
7. Dissecting the Built-In Self-Test ... 46
7.1. The SVF File ... 46
7.2. The EVF2 File ... 53
8. Putting It All Together ... 61
8.1. Demonstration of the Built-In Self-Test ... 63
8.2. Summary and Conclusions ... 63
9. Sample Files ... 65
9.1. BSDL Files ... 65
9.2. SVF File ... 68
9.3. EVF2 File ... 70
10. References ... 75

Acknowledgements

Permission to use screen capture images and listings was provided by Corelis, JTAG Technologies, and Flynn Systems Corporation. Engineers from these companies also provided valuable assistance and consulting in the development of the material presented in this design guide. Their help is gratefully acknowledged.

A Word about the Automatic Test Pattern Generator Tools

This design guide focuses on built-in self-test. The Joint Test Action Group (JTAG) Test Access Port (TAP) can be used for much more than built-in self-test, but this is beyond the scope of this design guide. For the built-in self-test example in this design guide, Automatic Test Pattern Generator (ATPG) tools from three vendors were used. The three tools were JTAG Technologies ProVision, Flynn Systems onTAP, and Corelis ScanExpress TPG.

Each ATPG tool vendor supplies test hardware designed to work with that vendor's ATPG tool. All three vendors provide their ATPG tools as part of a complete test system. These test systems are useful for manufacturing test, engineering debug, and any other areas where the JTAG TAP finds application beyond built-in self-test.

The procedures for using these tools described in this design guide focus on the use of the tools to produce 'static' Serial Vector Format (SVF) files. These files are generated prior to system deployment using an ATPG tool, which is external to the system to be tested. The files are then used for built-in self-test on multiple units of the system to be tested. This procedure generally produces limited diagnostic information. Extensive diagnostic information requires a more dynamic approach. All of the ATPG tool/test hardware systems are designed to enable extensive diagnostics by performing dynamic, flexible, and powerful board tests using the JTAG TAP. These diagnostic tests can be tailored by the ATPG systems "on the fly" to zero in on a detected fault and diagnose it. For most applications, the use of these tools for manufacturing test provides capabilities beyond those that can be reasonably implemented in a built-in self-test.

The design guide focuses on the use of these tools to produce 'static' SVF files for deployment on embedded test hardware for the purposes of built-in self-test. All of these tools can be used for and perform equally well for this purpose. The selection of an ATPG tool is a matter of personal preference and company history. Nothing in this design guide is intended to, or should be interpreted to suggest that one or the other of these tools is preferred for built-in self-test. They can all perform the job described in this design guide, and they can all do much more.

Discussion of the capabilities and merits of the various tools is beyond the scope of this design guide. For further information on the ATPG tools and the associated test systems, it is suggested that the reader contact the various tool vendors.

1. Introduction and Scope

1.1. About this Design Guide

This design guide provides a roadmap for implementing IEEE 1149.1 boundary scan functionality on a printed circuit board. The guide is divided into several major chapters:

- **Introduction and Scope** – This chapter provides a brief historical introduction to boundary scan techniques and describes the purpose and scope of the design guide itself.
- **A Brief Introduction to Boundary Scan** – National's SCANSTA family of boundary scan support devices is introduced in this chapter. This chapter also provides a high-level summary of IEEE 1149.1 operations.
- **Test System Description** – This chapter describes the system used as an example in the guide. The system includes hardware and software. It is meant to model, in a simplified way, an operational system that might include IEEE 1149.1 built-in self-test.
- **Hardware Design Considerations for Built-In Self-Test** – The hardware design considerations involved in implementing built-in self-test using the IEEE 1149.1 TAP are described. The conclusion of this chapter is that IEEE 1149.1 operation is very easy to implement from a hardware standpoint.
- **Development of a Boundary Scan Test Pattern** – This chapter describes the steps involved in using commercially-available Automatic Test Pattern Generator (ATPG) software to generate IEEE 1149.1 boundary scan test patterns for built-in self-test.
- **Embedded Vector Delivery Software** – The embedded vector delivery software to be implemented on the system controller is outlined in this chapter. The software provided by National Semiconductor for embedded vector delivery is straightforward to implement and flexible.
- **Dissecting the Built-In Self-Test** – This chapter takes a closer look at the mechanics of the IEEE 1149.1 operation. The "behind-the-scenes" look at boundary scan included in this chapter is meant to provide deeper insight into how boundary scan works.
- **Putting It All Together** – All of the design considerations presented earlier are summarized. In this chapter, an example of a boundary scan built-in self-test in operation is presented.
- **Sample Files** – The sample file listings provide complete listings of the data files required for the IEEE 1149.1 built-in self-test development and deployment.
- **References** – This section provides references used in the text and additional information.

Note: Active low signals are designated in this design guide by a trailing asterisk. For example, the active low asynchronous reset line is designated TRST*. Other conventions commonly used for this type of signal are an overbar, which is used in some of the figures in this design guide, and a trailing slash.

1.2. An Engineer's Historical Perspective

In the early days of printed circuit (PC) board design and development, PC boards were simple enough to be layed out by hand. 1/8 inch tape and pre-cut pad shapes were used to lay out boards with through-hole devices at 2x or 4x scale on sheets of Mylar. By the standards of the time, this process of cutting tape and placing pads with a #11 X-acto knife was state of the art.

Then, over the next twenty years, devices went from through-hole to surface mount, and then to advanced chip-scale packages and ball-grid arrays. PC boards went from one or two layers to fifteen, twenty, or more. Contact pitches got smaller as did the pads on the board. It became difficult even to touch a scope probe to a pad on a device for testing because the pad was smaller than the probe tip – often much smaller. On many traces the pads were underneath the device, or on an interior layer that wasn't accessible.

PC boards became denser and their functions became more sophisticated. Engineers could and did design much more functionality into much less space. Twenty boards with twenty integrated circuits each were replaced by one much smaller board with ten integrated circuits of greater functionality and higher cost. Testing these highly-integrated boards was more critical, because of the relatively higher cost of the sophisticated boards, and more difficult because of their small geometries and buried circuitry. PC board testing was a problem waiting for a creative solution.

The creative solution came about in the form of the IEEE 1149.1 boundary scan test standard[1], also known as JTAG (for Joint Test Action Group, the working group that originally formulated it). The standard is a set of design rules for integrated circuits (principally digital integrated circuits at first but now, increasingly, other integrated circuits as well) which is designed to facilitate board-level testing. The standard does this by specifying an auxiliary test port on each integrated circuit called the Test Access Port or TAP and protocols to address and use it. Through the TAP, many, most, or even all of the pins of an integrated circuit are accessible for testing. If a pin on a device is supposed to be connected by a board trace to a pin on another device, and if both pins are accessible for testing through the TAP, then the connection between them can be tested. And if many, or most, or all of the connections

on the board can be tested then many board failures can be detected, diagnosed, and perhaps even repaired early in the manufacturing process where test and debug are less expensive.

Since the JTAG standard was first formulated, new applications for it have proliferated. For example, JTAG is now widely used for in-system programming of programmable devices as described by IEEE Standard 1532[2]. JTAG is also used to provide built-in self-test capability for modern high-density PC boards. To assist in the deployment of these systems, National Semiconductor has developed a family of JTAG support devices. These include the SCANSTA101 JTAG System Test Access (STA) master[3], the SCANSTA111[4] and SCANSTA112[5] JTAG Scan Bridge multiplexers, and the SCANSTA476 analog voltage monitor[6]. These devices solve system problems that are difficult to attack by any other means.

1.3. Purpose of this Design Guide

Development of a useful boundary scan test is the classic journey of 1000 miles that begins with a single step. This design guide describes how to take that first step – and how to take the next several steps as well.

This design guide describes a case study: a simple board designed specifically to demonstrate boundary scan test. The design guide describes the board features enabling boundary scan test. As the board is described it will become apparent that, from a hardware perspective, the addition of boundary scan test to the board requires very little additional engineering beyond that required for the board's base functionality.

The design guide also describes the use of commercially-available Automatic Test Pattern Generator (ATPG) tools to develop test patterns for the demonstration board. The look and feel of each tool is unique, but the use models for all of them are very similar – similar enough so that a detailed demonstration with one tool will also be instructive to users of other tools.

Once the test patterns have been generated, the design guide will take the reader behind the scenes to describe what the test patterns are really meant to accomplish. The design guide will also take the reader through the process of converting the test pattern output to a form useful for deployment in an on-board test application, and will show how to convert it back into a human-readable form to gain further insight into what the built-in test will really do when it is deployed.

Finally, the design guide will describe one method of deploying the test pattern to the board, with the aim of providing a useful guide to the system engineer developing a similar built-in test for an operational board or system. It will describe how the test is conducted and what both success and (artificially-induced) failure look like. The design guide will conclude with some useful general conclusions and some suggestions on how to follow the example described here.

This design guide was written from the perspective of what a system designer would do in developing a boundary scan self-test. The objective was to show the reader what steps are required and where the pitfalls are so that the reader can avoid these when he/she develops a boundary scan-based board self-test.

2. A Brief Introduction to Boundary Scan

2.1. National's Boundary Scan Family

National manufactures a family of digital integrated circuits targeted at board- and system-level applications using the IEEE 1149.1 system test access port, also known as boundary scan. These devices include:

- SCANSTA101 Low-Voltage IEEE 1149.1 System Test Access (STA) Master
- SCANSTA111 Three-Port Addressable Multidrop IEEE 1149.1 JTAG Multiplexer
- SCANSTA112 Seven-Port Addressable Multidrop IEEE 1149.1 JTAG Multiplexer
- SCANSTA476 Eight-Input IEEE 1149.1 Analog Voltage Monitor

These devices enhance the functionality and ease-of-use of the IEEE 1149.1 system test access port. National's SCANSTA101 low-voltage IEEE 1149.1 STA master, for example, enables in-system IEEE 1149.1 boundary scan and in-system programming for programmable devices. Its primary function is to present to an on-board or external controller a simplified parallel interface to the serial IEEE 1149.1 boundary scan chain. The SCANSTA101 STA master is supported by National software that simplifies the interface to the on-board microcontroller and to the design tools that generate the boundary scan test vectors.

National's boundary scan support software includes:

- EVF Workbench which provides a graphical user interface for converting Serial Vector Format (SVF) files to Embedded Vector Format 2 (EVF2) files
- SVF2EVF which is the SVF to EVF2 compiler
- EVF2 vector delivery -- the suite of embedded functions that drive the SCANSTA101 STA master from the EVF2 file

National's SCANSTA111 and SCANSTA112 addressable multidrop IEEE 1149.1 STA multiplexers enable simplification of system designs by breaking long boundary scan chains into multiple, shorter chains. This capability is supported in Automatic Test Pattern Generator (ATPG) software from several major software vendors. These devices enable faster, easier-to-implement, and more robust in-system test access.

National's SCANSTA476 eight-input IEEE 1149.1 analog voltage monitor extends the digital IEEE 1149.1 into the analog domain. The SCANSTA476 monitor enables the monitoring of analog voltages for in-system test using protocols similar to those of the standard IEEE 1149.1 system test access port.

2.2. The IEEE 1149.1 Standard

The system Test Access Port (TAP) was defined in IEEE Standard 1149.1-1990[1 7]. As the name indicates, this standard was ratified in 1990, nearly two decades ago as of this writing. This standard and its subsequent updates defined a set of design rules, mostly meant to apply to integrated circuits, which were intended to facilitate board-level tests. Dense, multi-layer printed circuit boards with surface-mounted components, often on both sides, were becoming increasingly common. This made it difficult to ensure that a conventional board tester would be able to access all the required test points. The standard provides a remarkably ingenious solution to this problem. The cost of the ingenuity of the solution is that its implementation may be complex. This design guide describes how National's devices can be used to implement a boundary scan-based built-in self-test quickly and easily. These devices, and the software that supports them, are designed to reduce the complexity of the IEEE 1149.1 interface.

This design guide focuses on the IEEE 1149.1 system test access port and National's family of devices that support it. The design guide provides a roadmap for system designers implementing in-system boundary scan with National's IEEE 1149.1 support (SCAN) devices. It answers basic questions including:

"What can National's SCAN devices do for me?"
– Section 2.3.3. Use of the IEEE 1149.1 TAP

"How do National's SCAN devices work?"
– Section 3.1.1. PCI-1149.1/101 SCANSTA101 PC Card
– Section 3.1.3. SCANSTA111 Intermediate Board

"How do I implement National's SCAN devices in my board design?"
–Chapter 4. Hardware Design Considerations for Built-In Self-Test

"How do I generate the test vectors National's SCAN devices will use?"
–Chapter 5. Development of a Boundary Scan Test Pattern

"How do I convert the output of my Automatic Test Pattern Generator program to a format the National SCAN devices understand?"
– Section 5.6. Generation of an EVF2 File Using EVF Workbench

"How can I deliver the test vectors to the SCANSTA101 master in my system?"
– Chapter 6. Embedded Vector Delivery Software

"What is really going on 'under the hood' in the boundary scan process?"
– Chapter 7. Dissecting the Built-In Self-Test

"What should I expect if my board is working correctly? What if it has a manufacturing defect?"
– Section 8.1. Demonstration of the Built-In Self-Test

This design guide is intended for engineers designing new systems with National's SCAN devices and for engineers supporting systems that already include National's SCAN devices. The information contained herein will enable system designers to more easily utilize National's SCAN devices in their designs, leading to improved ease-of-use, acceleration of design cycles, improved manufacturability, and superior system designs.

2.3. Boundary Scan Basics

The history of and motivation for boundary scan testing techniques is a fascinating subject, and is well worth an investment of several hours of research. The references in Chapter 10 include excellent treatments of these subjects. They are recommended reading for anyone contemplating the use of JTAG.

This design guide, however, begins with the assumption that the reader has already made the decision to use the IEEE 1149.1 TAP for in-system boundary scan testing or programming of programmable logic devices. In this section a generic example system is described and demonstrated, along with where and how boundary scan can be applied. Following is a brief description of the IEEE 1149.1 TAP and what it can do.

2.3.1. Elements of the IEEE 1149.1 Test Access Port (TAP)

From a conceptual standpoint, the IEEE 1149.1 TAP consists of several functional units:

1. Four (or five) pins on each digital device in the system implementing the IEEE 1149.1 TAP. These pins form the test access port and they are separate from, and may not be shared with, any other functions of the device. The four required pins are Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI) and Test Data Out (TDO). The optional pin is an asynchronous, active low Test Reset (TRST*).
2. Digital circuitry which forms the TAP controller on each device implementing the IEEE 1149.1 TAP. The TAP controller is a finite state machine with functionality fully described in the standard.
3. An instruction register for the TAP on each device implementing the IEEE 1149.1 TAP. Conceptually, this register controls the behavior of the other registers in the device which are associated with the IEEE 1149.1 TAP.
4. A one-bit bypass shift register which can be inserted (by using the instruction register) between the TDI and TDO pins.
5. Boundary register cells between each pin of the device and the internal logic connected to the pin. These devices form a boundary register which can be inserted (by using the instruction register) between the TDI and TDO pins. This is the key element of the IEEE 1149.1 standard.
6. Other registers and control logic, some required and some optional.

The boundary register cells provide an alternate way to control all the outputs of the device as seen from the device pins. They also provide a way to monitor all the inputs of the device as seen from the device pins. This is like having the ability to probe every line connected to every device (or at least every device that implements the IEEE 1149.1 TAP) in the system. Clearly this is an extremely powerful test capability.

But this is not all that the TAP provides. The boundary register cells can also control the inputs of the device as seen from its core logic, and they can also monitor the outputs of the device as seen from its core logic. This provides both the ability to test any device implementing the IEEE 1149.1 TAP, and the ability to control the inputs of any device independent of the other circuitry on the board.

Finally, the standard for the TAP provides extensibility. It permits device manufacturers to use the IEEE 1149.1 TAP for other purposes. One common use for the IEEE 1149.1 TAP is in-system programming of programmable devices.

2.3.2 Architecture of the IEEE 1149.1 TAP

Seen from the viewpoint of the TDI and TDO pins, the IEEE 1149.1 TAP is a one-bit serial port. Data is clocked in to the TAP one bit at a time and clocked out one bit at a time. What happens between the TDI and TDO pins is controlled by an additional single control bit, the Test Mode Select (TMS). These three pins, along with the Test Clock (TCK), can be used to provide a remarkable range of behavior. The trick is

to determine what sequence of bits on the two input pins will produce the desired behavior in the system and what sequence of bits to look for on the output pin to determine the results of the desired test. Fortunately this problem is amenable to a significant degree of automation.

Most modern digital devices are designed using software tools that provide additional levels of abstraction between the desired behavior (often described using a hardware description language such as VHDL or Verilog) and the digital circuitry required to implement it (gates, flip-flops, latches, multiplexers, etc., usually many of them connected in a complex fashion). Similarly, the IEEE 1149.1 TAP was conceived with the intent to rely on software tools to generate the required digital sequences to produce the desired behavior in the system. This simplifies the job of the system designer since he or she does not need to take the design down to the lowest level of operation (long sequences of bits). Nonetheless, understanding this lowest level of operation will help the system designer use National's SCAN devices effectively.

A conceptual design of the IEEE 1149.1 TAP is shown in ***Figure 2-1***. The cloud labeled "Logic" represents all the internal logic of the device. The port labeled "Input" represents one of the input pins of the device and the port labeled "Output" represents one of the output pins. The flip-flops and multiplexers between the pins of the device and the internal logic represent a conceptual description of one cell in the boundary register.

Each cell in the boundary register holds one bit. The input to each cell can be driven by (1) an output from the previous cell or (2) for an input pin, the input to the device at the pin, or (3) for an output pin, the output from the device's internal logic. When the output of one cell drives the input to the next, the boundary register looks like a shift register. The contents of the boundary register can be unloaded in a parallel operation into the internal logic inputs to the device or into the output pins of the device. This arrangement provides a powerful mechanism for test access.

As seen in the figure, there are other registers that can be connected between TDI and TDO. The boundary register is made up of all the boundary cells, and it can be connected to the operational parts of the circuitry. Other registers such as the instruction register, the ID register, and the bypass register are not directly connected to the operational circuitry. The IEEE 1149.1 standard specifies that the TAP controller determines which register is connected between TDI and TDO and what data it shifts in and out of the TAP. The TAP controller is implemented as a 16-state finite state machine. A state diagram of the TAP controller is shown in ***Figure 2-2***.

The labels on the state transition arrows are the values asserted on the TMS line by a JTAG controller such as National's SCANSTA101 STA master. To understand the operation of the TAP controller, what happens when the TMS line is held high for five clock cycles (this is called a "five high TMS reset") should be considered. Start from any state in the state diagram

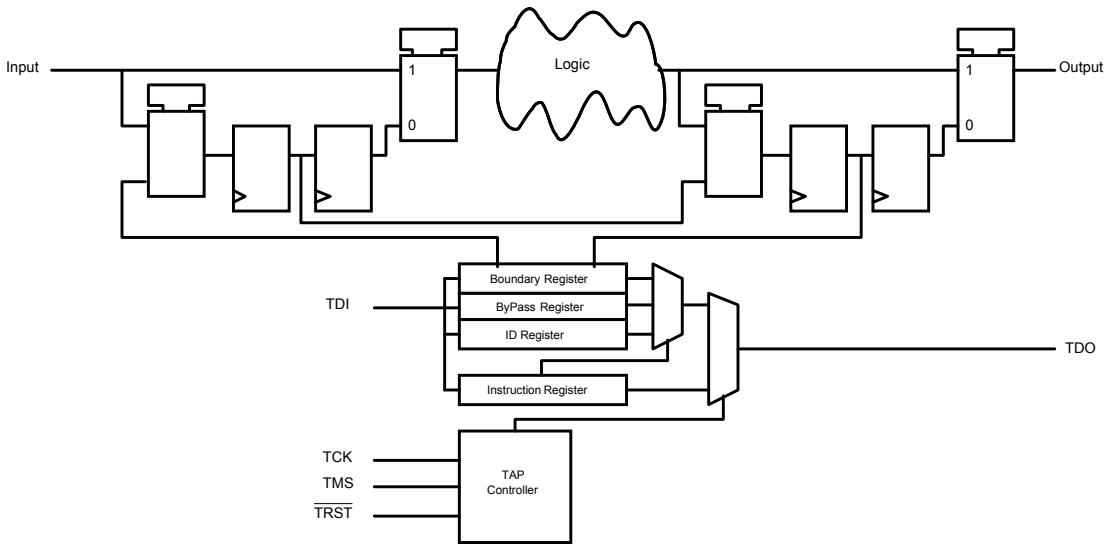

Figure 2-1. Architecture of the IEEE 1149.1 Test Access Port (TAP)

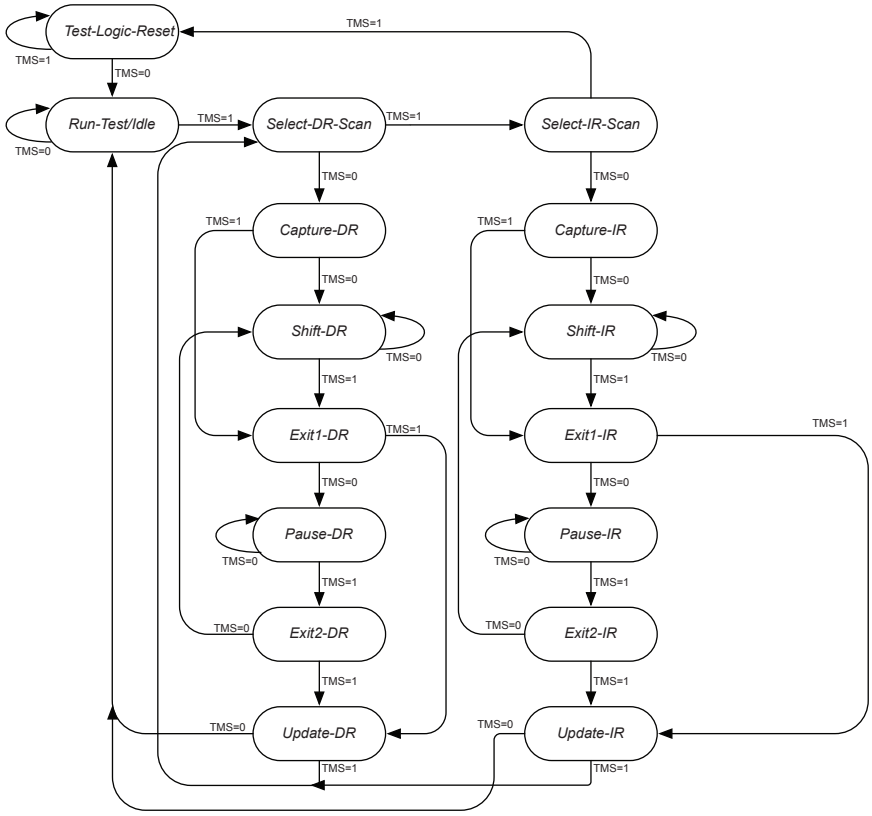

Figure 2-2. State Diagram of the IEEE 1149.1 TAP Controller

and follow five transitions with the TMS line held high. The intermediate states through which the TAP controller state machine passes depend upon where it starts, but after five transitions with the TMS line held high the state machine will always be in the Test-Logic-Reset (TLR) state, and it will stay there as long as the TMS line is held high.

There are four "stable" states in the above diagram. These are states in which the TAP controller can remain for as many successive TCK cycles as desired by holding the TMS line at a given value. In these "stable" states, the only operations that occur in the TAP are operations that have been previously set up and activated. These states are Test-Logic-Reset, Run-Test/Idle, Pause-Data-Register (Pause-DR), and Pause-Instruction-Register (Pause-IR). If the TAP controller is in any other state, it will either transition to a different state or it will shift data into the data or instruction register on the next TCK rising edge.

The Shift-Instruction-Register and Shift-Data-Register states are stable in a sense, in that the state machine can remain in these states as long as the TMS line is held low. When the TAP controller is in one of these states however, it is actively loading some register and the overall condition of the system is changing. For this reason, these states are not considered "stable" in the same sense that the four states previously described are considered stable.

Up to now only the TAP controller for a single device in the system implementing the IEEE 1149.1 TAP has been considered. For the case where there is only one boundary scan chain (i.e., where National's SCANSTA111 multiplexer or SCANSTA112 multiplexer is not used in the scan chain, or where these devices are used, but the multiple scan chains are all tied together), the TAP controllers for all the devices move from one state to the next in unison. If all the TAP controllers start out in the same state, they all remain in the same state. So it is really only necessary to consider a single TAP controller state for the entire scan chain.

With this state transition mechanism and the ability to shift data in and out of various shift registers, the IEEE 1149.1 TAP can produce a wide variety of complex behaviors. This provides a very powerful mechanism for in-system testing, programming, and diagnosis.

A Brief Introduction to Boundary Scan

Use of the IEEE 1149.1 TAP consists, in essence, of driving the TMS and TDI lines with the correct bit sequences to accomplish the desired functions, and monitoring the TDO line for the desired responses. These three lines and the test clock essentially comprise the IEEE 1149.1 boundary scan standard.

2.3.3. Use of the IEEE 1149.1 TAP

Given the simplicity of the architecture of the IEEE 1149.1 TAP, the question arises: What is its utility in a system? The answer is illustrated in the digital system example in ***Figure 2-3***. The figure shows several devices implementing the IEEE 1149.1 TAP interconnected in an operational system. The desired functionality of the system is embodied in the interconnections between the input and output pins of the various devices. Even though the IEEE 1149.1 TAP pins are connected between the devices on the board, these connections could all, in principle, be removed without affecting the desired functionality of the system. This is a key point. The IEEE 1149.1 TAP is meant to function independently of what the system is otherwise designed to do.

The example of ***Figure 2-3*** is deliberately left generic and shown not to be greatly complex. Conceptually, however, a considerably more complex system could be represented in the same way as the system of ***Figure 2-3***. Consider the connections labeled A and B in ***Figure 2-3***. These connections might be traces on a printed circuit board, vias, wires, connectors, cables, or some combination of all of these.

Suppose these two connections were shorted together because of some manufacturing defect in a particular unit. In this case, it is very likely that the unit would not work correctly, at least some of the time. Obviously the manufacturer of the system would prefer to identify and repair or discard the defective unit before it was shipped to a customer. This becomes even more critical as the system becomes more complex and, probably, more expensive. But it also becomes more difficult.

In a complex system, functional testing may not identify the problem with a faulty unit. Complex systems exhibit complex behaviors. (Simple systems can too, but complex systems almost always do. A complex system that exhibits simple behavior is likely to be replaced by a simpler system.) Testing a complex system to identify a manufacturing defect like the one described above by observing its normal functionality would require exercising enough of its complex behavior to ensure that some anomalous observation would occur should there be a manufacturing defect in the system. This becomes progressively more difficult, expensive, and time-consuming as the behavior of the system becomes progressively more complex. In addition, even if the presence of a given defect could be detected by observing some anomalous behavior in the system's normal functional environment, it is unlikely that the exact location and nature of the defect could be identified in this way. Many defects might produce the same anomalous observed behavior. As the system becomes more complex, just putting it through its normal operational paces becomes a less satisfactory method of testing and diagnosing faults.

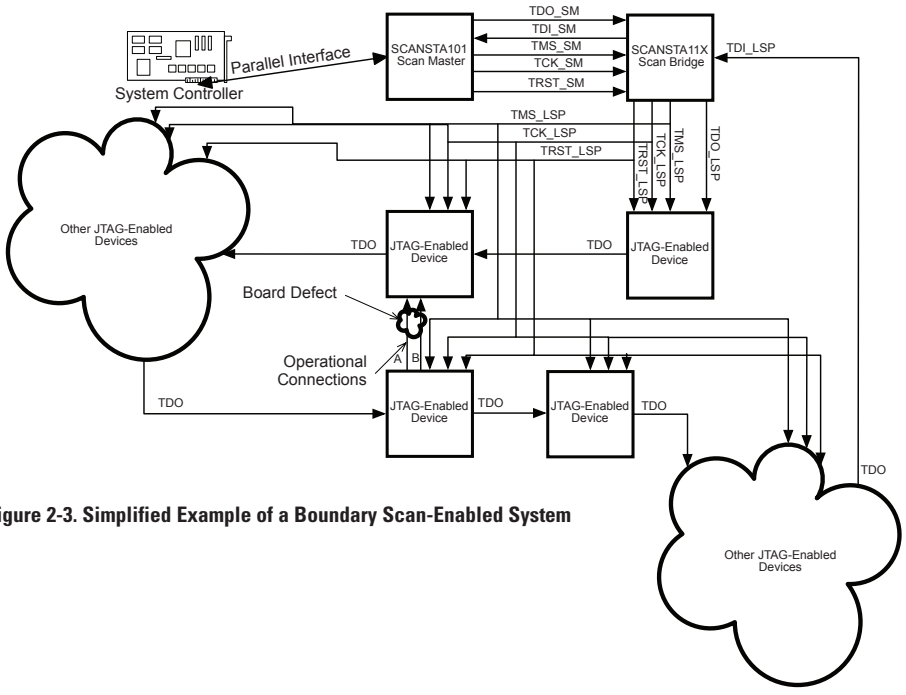

Figure 2-3. Simplified Example of a Boundary Scan-Enabled System

What is really required to identify a manufacturing defect like that previously described is the ability to impose various signals at the driving nodes of connections A and B and to observe the signals received at the receiving nodes of those connections. If there is a defect in the connection, the received signal will be different from the driven signal. Also, if it is known which nodes were being driven and where the signals were being received, then it is also clear exactly which paths to examine in order to locate the defect.

This is what the IEEE 1149.1 TAP can do.

In principle, using the IEEE 1149.1 TAP, all of the driving boundary cells in the entire design (the ones that implement IEEE 1149.1 TAP, anyway) can be connected into a single, large shift register, and a desired data pattern can be shifted into that shift register. All of the receiving boundary cells can be connected in the same way into a large shift register (in practice, it's the same shift register for input and output boundary cells). The boundary cells can then be used to drive the desired data pattern onto the connections on the board and to receive the resulting pattern from the connections on the board. Then the received data can be shifted out of the receiving boundary cells and compared to the data that is expected.

If this is done for several different data patterns it is possible, in principle, to identify and isolate a large percentage of the possible faults in a system. Even in realistic, complex systems, fault coverage (the probability of identifying a defect this way) can be computed a-priori, and can approach 100%. If the system has device pins that are bidirectional (input and output), the pins can be exercised in both modes using different data patterns.

Problems can be diagnosed in the internal functionality of devices on the board by driving their input pins and observing their output pins using the boundary cells. Semiconductor devices are often tested this way in manufacturing. A device that implements the IEEE 1149.1 TAP can be tested after it has been installed in its target system application.

Programmable logic devices in the system also can be programmed using the IEEE 1149.1 TAP. The IEEE 1149.1 TAP provides a secondary I/O port to these devices and the IEEE 1149.1 standard is written to permit extensions of the standard to applications such as in-system programming.

All of this IEEE 1149.1 functionality can be implemented in the system itself, enabling self-test, health monitoring, and in-system programming updates. When this capability is integrated into the system itself, it requires a control mechanism of some sort and enough memory to store the required test patterns and expected data for comparison.

This is the application for which the SCANSTA101 STA master was designed and it is the subject of the present design guide. This will be described in more detail in the remainder of this guide.

2.4. Summary and Conclusions

As described in this introductory chapter, the IEEE 1149.1 TAP provides an independent mechanism for accessing the inputs and outputs of a device (at its "boundary"). In this way it enables precise and extensive testing for manufacturing defects in a system. It also permits extensive testing of individual devices either in a system or in isolation, as in a manufacturing test. It also permits in-system programming of programmable devices. These capabilities may be utilized by external equipment such as test systems or device programmers, or by devices within the system itself.

So far all the descriptions of the operation of the IEEE 1149.1 TAP have described low-level functionality; namely, functionality at the level of sequences of bits. This is analogous to the machine code that describes a computer program at the lowest level. The memory containing a computer program really just contains a sequence of bits. The meaning in the bit sequences is expressed when the computer retrieves them from memory and applies them to its (complex) internal structure.

Human beings could monitor and examine the bit sequences applied to the TDI and TMS pins of an IEEE 1149.1 TAP in order to understand the functions being performed by the TAP just as they might examine the bit sequences that make up a computer program. Extracting meaning from these bit sequences would be a difficult proposition in the general case. So one would do what human beings usually do in such cases – look for patterns and abstract common features from the low-level description represented by the bit sequences. This abstraction is inherent in the architecture of National's family of SCAN devices. How that abstraction is accomplished will be examined in this design guide.

3. Test System Description

3.1. The SCANSTAEVK Demonstration Kit

National's family of boundary scan support devices provides a valuable built-in self-test capability for system implementations. National has developed a demonstration kit to help system designers evaluate the capability provided by this family of products.

The demonstration kit includes a target board with two National boundary scan-enabled devices, a serializer and a deserializer, with relatively simple connections between them. It also includes a backplane with multiple JTAG port connections to accommodate multiple target boards and a set of intermediate boards for introducing the SCANSTA111 and SCANSTA112 JTAG multiplexers between the backplane and the target board. A PC-resident board with the SCANSTA101 STA master device for driving the JTAG ports completes the demonstration kit. In conjunction with the supplied, PC-based, vector delivery software, this demonstration kit can serve as a simplified model of a boundary scan-enabled operational system.

3.1.1. PCI-1149.1/101 SCANSTA101 PC Card

In an operational system implementing built-in self-test, a system controller would deploy test vectors to the boundary scan chain (or chains) by communicating, using simple parallel protocols, with one or more SCANSTA101 STA master devices. In the demonstration kit, the role of the system controller is emulated by a personal computer (PC). That the performance and characteristics of the PC are not important for the demonstration is one of the features that the demonstration kit is meant to illustrate.

On the internal PCI bus of the PC in the demonstration system, which is running Windows, a Corelis PCI-1149.1/101 boundary scan controller is installed. A photograph of this device is shown in ***Figure 3-1***. There are two National SCANSTA101 STA master devices on the PCI-1149.1/101.

The PC and the PCI-1149.1/101 boundary scan controller emulate the system controller and the SCANSTA101 STA master device in an operational system, but clearly an operational system would require much more limited capability to provide the built-in self-test function. The PC-based emulation subsystem used with the demonstration kit includes a GUI-driven vector delivery and evaluation software tool which communicates with the PCI-1149.1/101 using a low-level driver library supplied by Corelis. The driver library primarily provides simple functionality for reading and writing registers in the SCANSTA101 devices, just as it would be implemented in an operational system.

In an operational system, the system controller would, when commanded to perform a system self-test, communicate with the in-system SCANSTA101 STA master. In such a system, all the system controller would be required to do is perform a sequence of register reads and writes via a 16-bit parallel data bus and an associated 5-bit register address bus to the SCANSTA101 STA master device. The sequence of register reads and writes would be stored on board in a compact binary format called Embedded Vector Format 2 (EVF2). An embedded software function, provided by National in source code form, would provide the interface between the EVF2 format and the SCANSTA101 STA master device. As will be demonstrated in this design guide, the PC and the PCI-1149.1/101 boundary scan controller emulate this functionality in an instructive manner.

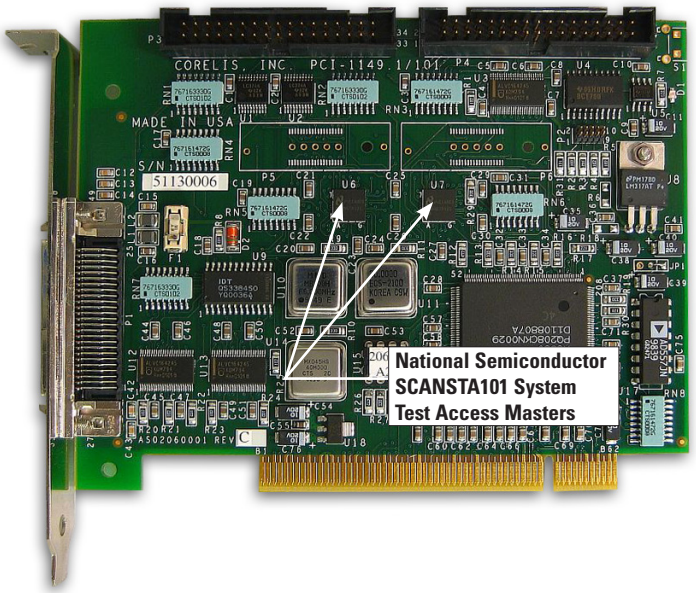

Figure 3-1. The Corelis PCI-1149.1/101 JTAG Controller Card with Multiple National SCANSTA101 STA Master Devices

3.1.2. SCANSTAEVK Demonstration Kit Backplane

An operational system might include a backplane designed to provide connections between the various functional boards in the system. In such an operational system, the system controller, and perhaps the SCANSTA101 STA master device, might be located on a master board and might test the other boards in the system by communicating with them over the backplane. Such a backplane would have multiple connections for various functional boards, and each such connection would include lines for the JTAG TAP. The SCANSTAEVK backplane emulates this functionality.

A photograph of the SCANSTAEVK backplane is shown in ***Figure 3-2***. A schematic of the backplane is shown in ***Figure 3-3***. This backplane is intended to demonstrate JTAG functionality. Accordingly, only the JTAG TAP lines are carried across the backplane, through the target card connectors, and to the target cards. In an operational system, signals associated with the primary function of the system would also be transported across the backplane and through the target card connectors. The SCANSTAEVK backplane, however, provides a realistic platform for testing the JTAG functionality. It is a key point of the IEEE 1149.1 JTAG standard that the TAP implemented in a device is independent of the primary function of the device. A backplane that does nothing but transport TAP signals is a reasonable vehicle for the development of a boundary scan built-in test application.

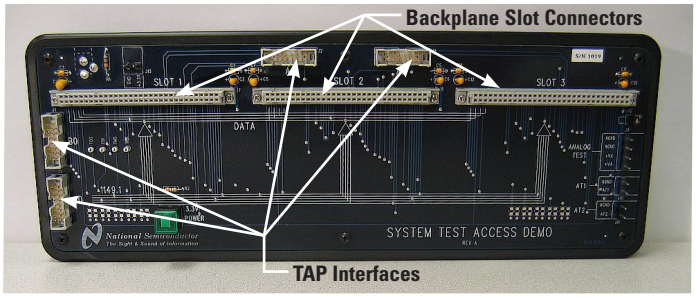

Figure 3-2. SCANSTAEVK Demonstration System Backplane with Three Target Card Connectors and Multiple JTAG Controller Ports

As can be seen in the schematic of ***Figure 3-3***, there are multiple TAP interface connectors on the board for TAPs labeled A0, A1, A2, and B0. The Serializer/Deserializer (SerDes) target boards have multiple TAP connections which can be configured by jumpers on the board. When these target boards are used with a JTAG multiplexer card, such as the one carrying the SCANSTA111 multiplexer, the active JTAG port on the target card is selected through the multiplexer. When a multiplexer is not used, all the scan ports on the target card can still be exercised by using different TAP interface connectors.

Associated with the A0 and A1 TAPs are signals labeled, for example, A0_FLASH* and A0_RDY/BSY*. These are auxiliary signals passed through the multiplexer when it is used. They are intended as general-purpose I/Os, each associated with either a single local scan port from the multiplexer or with the backplane TAP connector. These I/Os are not used in the SCANSTAEVK demonstration system described in this design guide.

3.1.3. SCANSTA111 Intermediate Board

Inserted between the backplane connector and the target board in the demonstration system is an intermediate card carrying a SCANSTA111 Scan Bridge JTAG multiplexer. The card has a plug that fits the backplane connector and a socket with the same connection configuration as the backplane connector into which one of the target boards may be inserted. The Scan Bridge card intercepts the input A0 TAP from the backplane and routes it to the SCANSTA111 multiplexer. The outputs of the multiplexer are then routed to the A0, A1, and A2 TAP connections on the target board connector.

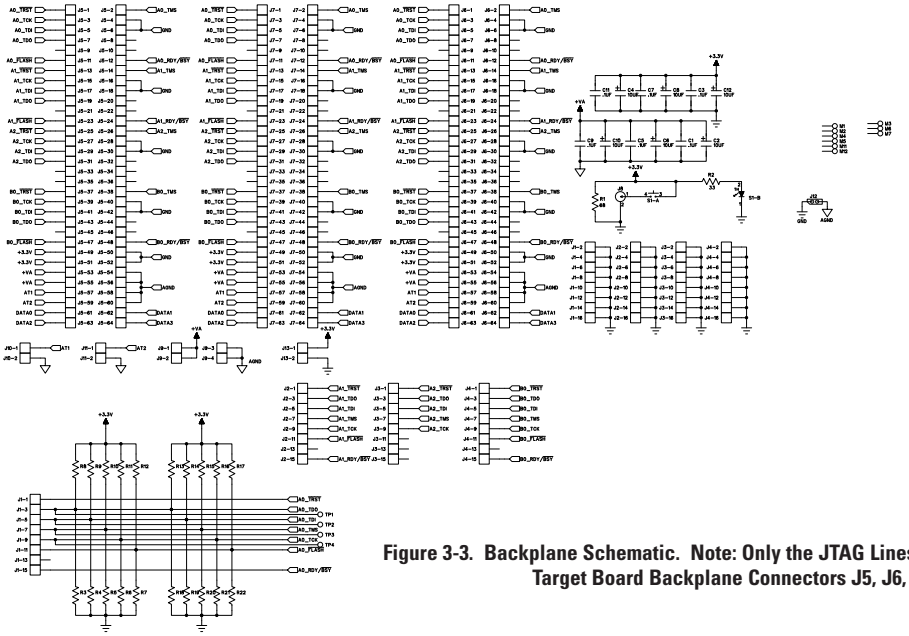

Figure 3-3. Backplane Schematic. Note: Only the JTAG Lines are Carried through the Target Board Backplane Connectors J5, J6, and J7.

Test System Description

A photograph of the intermediate Scan Bridge multiplexer card is shown in ***Figure 3-4***. A schematic of the card is shown in ***Figure 3-5***. On this card, J1 is the backplane connector. It makes connection only to TAPs A0 and B0. TAP A0 is routed on the card to the backplane master TAP of the SCANSTA111 multiplexer. B0 is routed directly to the output connector J2 for use in cases where it is desired to bypass the SCANSTA111 multiplexer.

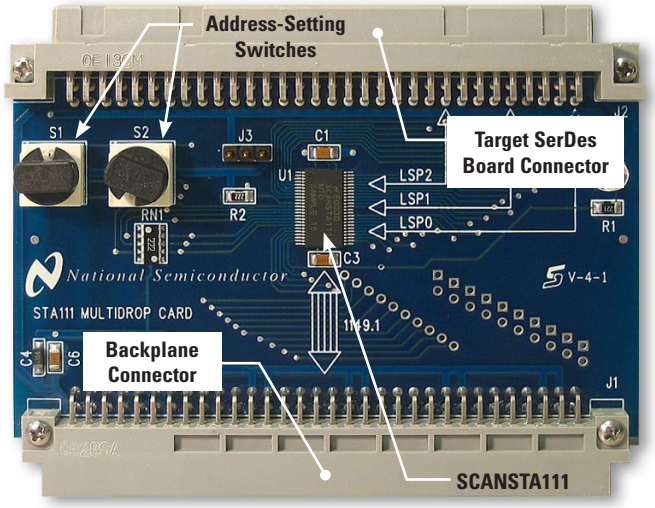

Figure 3-4. Intermediate Scan Bridge Multiplexer Card. Note the Address-Setting Switches on the Card and the Pass-Through Connector Arrangement.

The address of the SCANSTA111 multiplexer itself is set by switches S1 and S2. Selection of the SCANSTA111 multiplexer is accomplished by scanning an address matching the switch settings into the SCANSTA111 multiplexer's instruction register via the JTAG TAP.

Scan ports A0, A1, and A2 on the output connector, J2, are connected to the local scan port outputs of the SCANSTA111 multiplexer. The SCANSTA111 output port is selected by writing to the registers of the SCANSTA111 multiplexer through its JTAG master port, A0.

The selected local scan port of the SCANSTA111 multiplexer is routed out to the target card through the output connector. The connections on the output connector match those on the backplane. The target card can be connected directly to the backplane or to the output connector of the intermediate Scan Bridge card with no effect on the TAP operation of the target card. The software and the SCANSTA101 STA master must be aware of the SCANSTA111 multiplexer in order to control it and to account for the additional 1-bit delays it introduces in the TAP signaling, but the target card does not need to be aware of the presence of the SCANSTA111 multiplexer. As far as the target card is concerned, the Scan Bridge card is transparent.

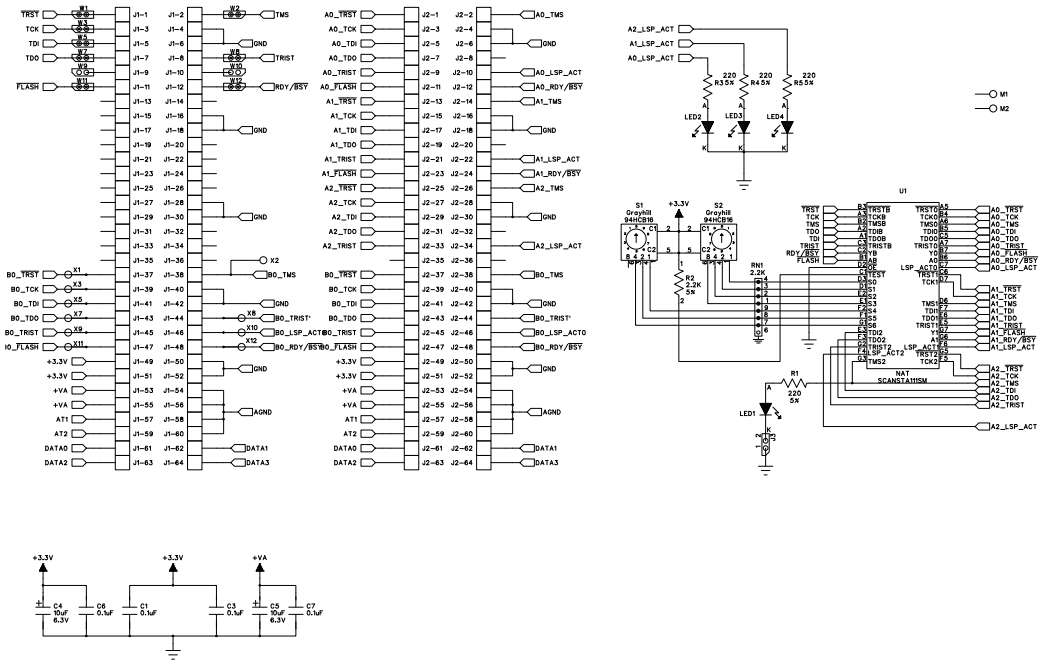

Figure 3-5. Schematic of the Intermediate Scan Bridge Multiplexer Card

3.1.4. Target SerDes Board

In the SCANSTAEVK demonstration system, the target SerDes board models the board to be tested using a built-in self-test in an operational system. The board includes National's serializer/deserializer pair, the SCAN921023/SCAN921224[8], both of which implement boundary scan. The SCAN921023 is an embedded clock 10-bit Low-Voltage Differential Signaling (LVDS) serializer. All of its digital inputs and outputs, including its primary differential LVDS output, are equipped with boundary scan cells. Similarly, the SCAN921224 is an embedded clock 10-bit LVDS deserializer designed to work with the SCAN921023 serializer. It, too, is equipped with boundary scan cells on all of its inputs and outputs including its primary differential LVDS input.

A photograph of the target SerDes board is shown in ***Figure 3-6***. The two-page schematic for the board is shown in ***Figure 3-7*** and ***Figure 3-8***. Reviewing the schematic will provide an indication of what could, in principle, be tested on this board if designing a built-in self-test.

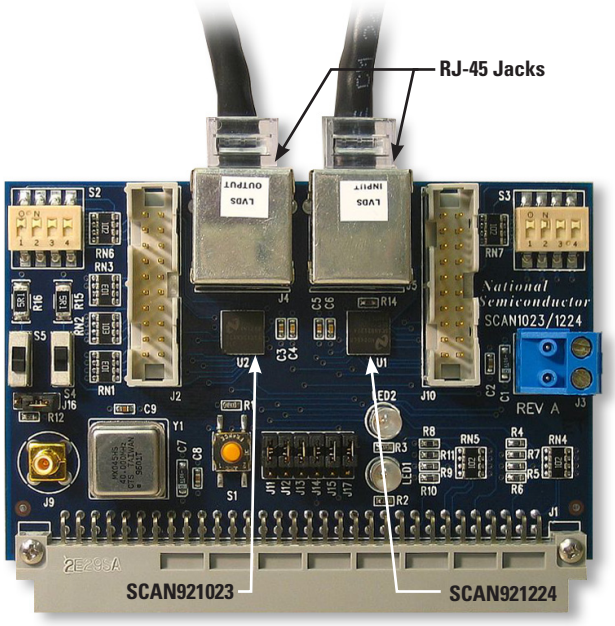

Figure 3-6. Target SerDes Board. The SCAN921023 Serializer and SCAN921224 Deserializer on this Board implement the JTAG Boundary Scan TAP.

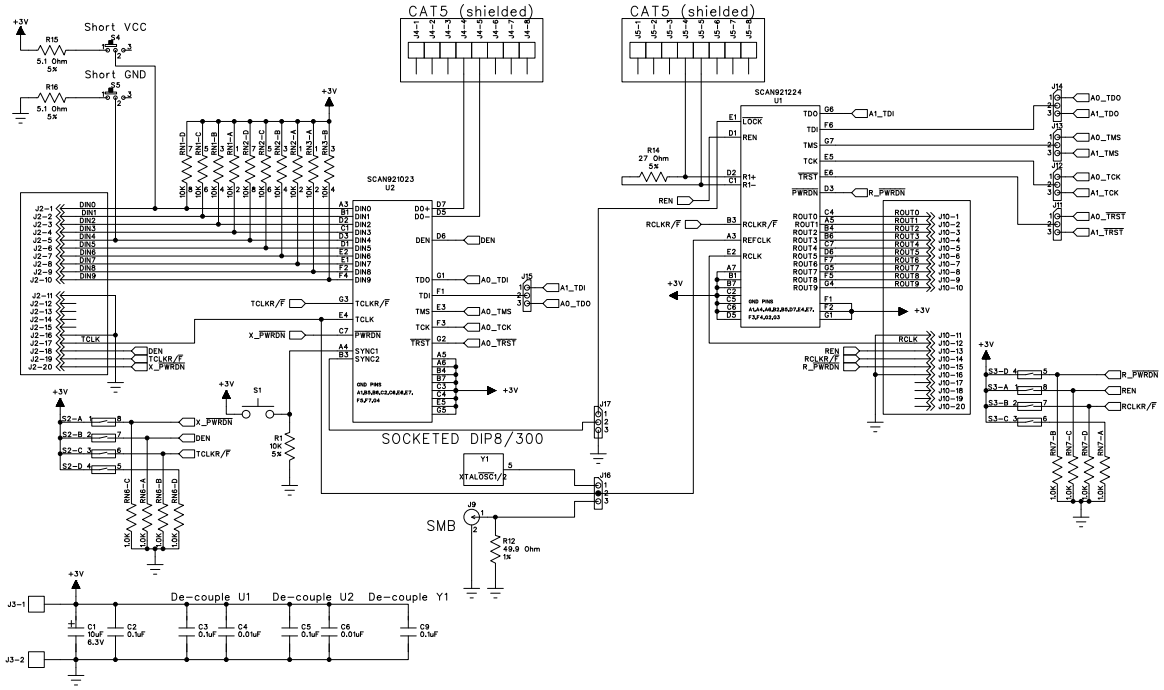

Figure 3-7. Target SerDes Board Schematic (Part 1). The Boundary Scan-Enabled Devices are the SCAN921023 Serializer and the SCAN921224 Deserializer.

Test System Description

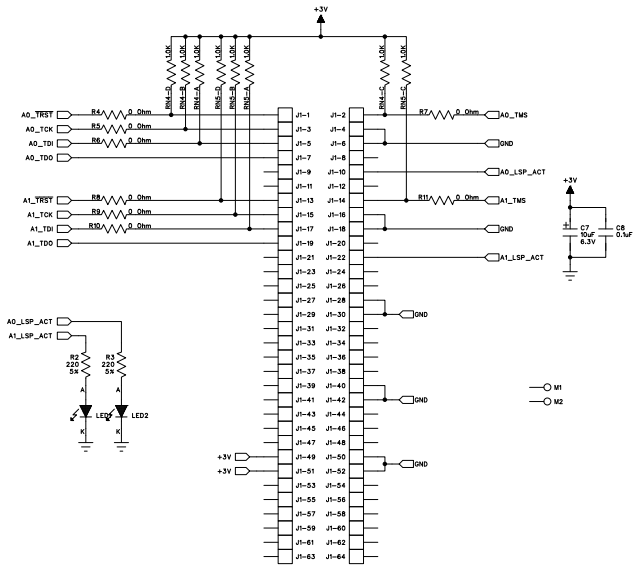

Figure 3-8. Target SerDes Board (Part 2). This Part shows the Backplane Connector.

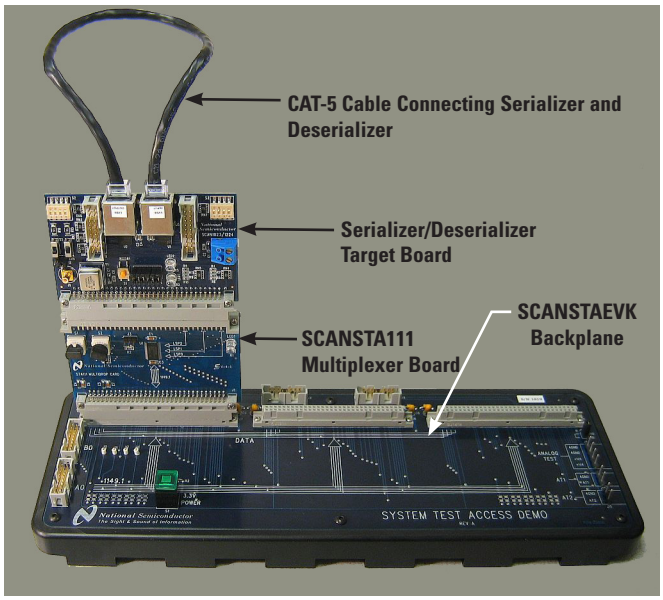

Figure 3-9. Complete Stackup for the SCANSTAEVK Demonstration Platform. This Stackup includes the Backplane, the Intermediate SCANSTA111 Multiplexer Board, and the SerDes Target Board. Note the CAT-5 Cable connecting the Serializer and Deserializer on the Target Board.

First, the parallel digital inputs on the SCAN921023 serializer, DIN0:DIN9, should be considered. If a data source isn't connected on connector J2, these inputs are all pulled up to +3V through RN1, RN2, and RN3. So detecting static high-input levels on these inputs to the SCAN921023 device should be possible, and, if detection is unsuccessful, that will indicate a fault in one of the resistor arrays or in one of the input lines. It is possible to introduce a fault deliberately by changing the setting of S5, and it should be possible to detect that deliberately-introduced fault.

Both the SCAN921023 serializer and the SCAN921224 deserializer have three inputs tied to static high levels by switches S2 and S3. The inputs are a PWRDN* input, an enable input, and a rising/falling edge clock selection input. All three are held high in normal operation, so it should be possible to detect static high levels on these input pins. If the settings of these switches are changed, it should be possible to detect the fault that this introduces.

The Sync1 input of the SCAN921023 serializer is tied low through a pull-down resistor unless a fault is introduced by depressing momentary switch S1. If this switch is depressed, detection of the fault should be possible.

The LOCK* output of the SCAN921224 deserializer can be connected by jumper J17 to the Sync2 input of the SCAN921023 serializer. If this connection is made, then it should be possible to drive this line from an output scan cell on the SCAN921224 deserializer and receive the signal driven at the SCAN921023 serializer. If the jumper is removed, a fault will be introduced (if looking for the presence of the jumper) that should be detectable.

Finally, a fault can be introduced in the connection between the serializer and deserializer on the SerDes target board. The SCAN921023 serializer has a differential LVDS output. The SCAN921224 deserializer has a differential LVDS input. In an operational system, the function of these two devices is to provide a two-wire serial data path between distant points in the system. Parallel data goes in to the SCAN921023 serializer and comes out as serial data. The serial data is routed to the SCAN921224 deserializer where it is recovered and output to the receiving system as parallel data. A serializer/deserializer pair in a system is used to reduce the number of connections needed to transmit data between distant points in the system.

In normal operation the differential LVDS output of the SCAN921023 serializer (DO+ and DO-) is connected through a cable to the differential LVDS input of the SCAN921224 deserializer (RI+ and RI-). In the SCANSTAEVK demonstration

kit, this connection is made by means of a CAT-5 cable with RJ-45 connectors.

The differential LVDS output and input pins on the SCAN921023 serializer and SCAN921224 deserializer are equipped with differential boundary scan output and input cells, respectively. If a test pattern is generated that tests this connection between the two devices and then the cable is removed, detection of the resulting fault should be possible.

3.1.5. Demonstration Kit Summary

If the SerDes target board of the SCANSTAEVK demonstration kit is used to model an operational system board then it is possible, by inspecting the schematic, to get an indication of what elements of the board structure are possible to test. In the case of this board, there is not much that can be tested - some static inputs to the two boundary scan-enabled devices and a couple of interconnects between them, including one differential LVDS interconnect.

Even though this model is highly simplified in comparison to any real operational board, it does capture many instructive features of the boundary scan built-in self-test problem. It also has the advantage that, since it is quite simple, it may be possible to trace the development of the boundary scan test procedures and thereby gain some insight into how these tests are developed and how the test development software works.

The SCANSTAEVK demonstration kit includes the SCANSTA101 STA master JTAG controller and the SCANSTA111 Scan Bridge multiplexer. Using the SCANSTAEVK demonstration kit as a model for an operational system will allow better understanding of the operation of these devices as well.

3.2. Test Development and Deployment Software

Designing and building the hardware to support built-in self test, and hooking up the test access ports properly, is only the first step in developing a built-in self-test. The SCANSTAEVK demonstration kit described previously was designed by National to illustrate the operation of National's SCAN family of devices. It should be noted, however, that if such a system were designed from scratch, it would be necessary to know very little about the boundary scan operation to take the design to this point. All that was needed was to connect the TAP properly between the boundary scan-enabled devices.

It would not have been necessary to know anything about the boundary scan operation in order to use the SCANSTA111 Scan Bridge multiplexer either. The design was simply a matter of connecting the backplane TAP and one or more of the local scan ports of the device properly. Also, it was not necessary to know anything about the boundary scan operation to implement the SCANSTA101 STA master either. It was only necessary that the SCANSTA101 STA master was connected to the correct address and data lines on the parallel processor interface side and to the TAP on the serial scan interface side.

This is an important point. What this means is that the hardware for a boundary scan-enabled system can be designed and fabricated well before any details of the boundary scan operation and the software required to implement it are known. The interfaces to the boundary scan-enabled devices, specified in IEEE 1149.1, and the interfaces to National's family of boundary scan support devices are all sufficiently well described and specified so that the hardware design and the software design can proceed almost independently; and the hardware can proceed first, which is good, because it probably takes longer to fabricate the hardware than to generate a first cut at the software (although it probably takes longer in the end to get the software fully debugged than it does to get the hardware working).

In the modeling exercise, however, the point has been reached where the software can no longer be ignored. The system can, in principle, perform a built-in self-test. So the question becomes: What is needed in order to implement this test?

3.2.1. Automatic Test Pattern Generation

All of the board faults have been described that could, in principle, be detected using a boundary scan built-in self-test. But how can these faults be detected in practice? What is needed is to scan in an appropriate pattern of bits into the boundary registers of the JTAG-enabled devices so that the outputs are correctly set, capture the pattern of bits at the inputs of these devices, and compare this to the expected data pattern. Conceptually it will probably take more than one cycle of setting the outputs and measuring the inputs to be sure that all the kinds of faults are detected that should be possible to detect. And this is a very simple board. How much more complicated will the process become when it is used on a more complex operational board?

The answer to these questions is an Automated Test Pattern Generator (ATPG) software tool. These tools are made by various manufacturers including Corelis, Flynn Systems, JTAG Technologies, Asset Intertech, and others. The tools are all different in their look and feel but all do the same job; namely, they generate test patterns that can be applied to the JTAG

TAP to detect all or most of the faults that can, in principle, be detected on a JTAG-enabled board.

An example of generating test patterns with these tools will be discussed in Chapter 5, but for now it is sufficient to say that the next step is to take a description of the board (a netlist), descriptions of the boundary scan-enabled components on the board (Boundary Scan Description Language or BSDL files), and some additional information like jumper settings and cable connections which are not on the netlist, and to use one of these tools to generate a set of test vectors for the board.

The ultimate output of this process is a Serial Vector Format (SVF) file, which describes the operations to be performed by the TAP in a human-readable format. Once the SVF file is created, the next step is to convert the SVF file to an Embedded Vector Format 2 (EVF2) file which can be deployed to the SCANSTA101 STA master.

The SVF file produced for the target SerDes board will be discussed in some detail in Chapter 7. For now, however, it is important to note that the first software-intensive step in the process is to generate a SVF file using a third-party ATPG tool.

3.2.2. Conversion to EVF2 Format

A SVF file contains instructions like STATE, which directs the TAP state machine to transition to a specified state, Scan Data Register (SDR), which scans data into and out of the TAP, Scan Instruction Register (SIR), which scans an instruction sequence into and out of the TAP, and other similar instructions. These are descriptive and complete, but the SCANSTA101 STA master requires instructions interpreted as a sequence of register reads and writes. This is the purpose of the EVF2 format.

The EVF2 format is a binary format consisting of data records that describe what data is to be written to what register in the SCANSTA101 STA master in order to accomplish each of the test sequences described in a SVF file. The binary format means that the EVF2 file consumes minimal storage in the embedded system's memory, an important feature for built-in self-test applications. National provides source code to be embedded in the system processor's code (actually this is just one C function) that interprets the EVF2 file records and sends out the appropriate address and data bits to set up the SCANSTA101 STA master.

The conversion to EVF2 format is accomplished by a program supplied by National called EVF Workbench. A screen shot of the main window is shown in ***Figure 3-10***. An example of using the simple interface of EVF Workbench will be highlighted in Chapter 5. For now, however, the next step after producing the SVF file is to convert it to an EVF2 file using EVF Workbench.

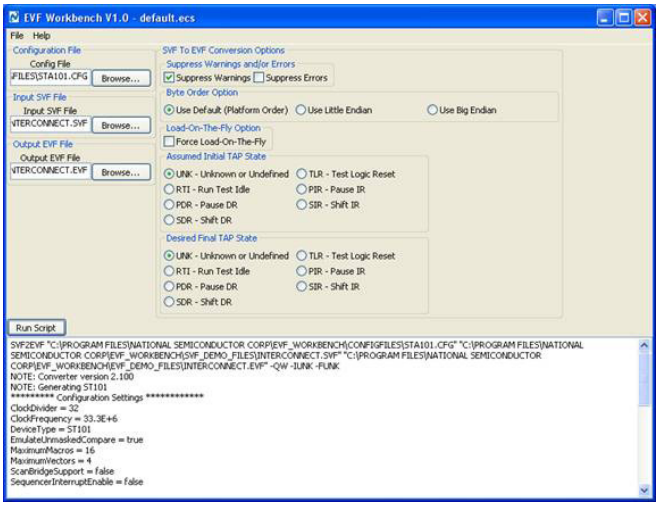

Figure 3-10. Main Window of EVF Workbench

After the file is converted to EVF2 format, it is possible to convert it to a human-readable representation of the EVF2 format. Normally this is neither required nor useful, but, in this case, it will provide some additional insight into how the SCANSTA101 STA master controls the self-test. A decompiled EVF2 file will be examined in Chapter 7.

3.2.3. Embedded Platform Software

Once these operations are complete, an EVF2 file is produced that details the register reads and writes required for the SCANSTA101 STA master to perform the desired built-in self-test. There must be some system processing power somewhere (a system controller, a DSP engine, a FPGA, or something else) that can drive the address and data lines of the SCANSTA101 STA master to perform the desired built-in self-test.

For systems with an embedded controller programmed in C or C++, National provides a function library that implements a simple Application Programming Interface (API) for delivery of the EVF2 file. It consists of a single user-called function and some additional glue functions to implement it. Following is the function's argument list.

```
int EVF2VectorDelivery(int (*pfGetData)(void *,void *,void *,size_t,size_t),
                       int (*pfErrorHndlr)(const char *,void *,void *),
                       int (*pfFailHndlr)(unsigned long,void *,void *),
                       int (*pfUserExecHndlr)(const char *,const char *[],void *,void *),
                       void (*pfRegTraceHndlr)(void *,void *,unsigned short,unsigned long,
                              unsigned long),
                       void (*pfDebugHndlr)(void *,void *,unsigned long,unsigned long,unsigned long,
                                    unsigned long,unsigned long),
                       void (*pfPollingHndlr)(void *,void *,unsigned long),
                       unsigned long  dwTimeoutMs,
                       unsigned long  dwLatencyMs,
                       void *pvUser1,void *pvUser2)
```

Most of the arguments to this function are function pointers to user-supplied, application-specific functions that handle the low-level operations of reading from memory and writing to the parallel address and data buses. This source code is added to the embedded processor code. When this function is called, at the appropriate time, it uses the passed function pointers to deliver the EVF2 file to the parallel port connected to the SCANSTA101 STA master.

In the demonstration system, this function is implemented in the PC software that controls the PCI-1149.1/101 card with the SCANSTA101 STA master on it. This implementation will be discussed in Chapter 6 and some analogies will be drawn between this implementation and a typical embedded application.

3.2.4. ScanVec

ScanVec is the PC application that reads an EVF2 file (from disk) and delivers the vector information to the SCANSTA101 STA master on the Corelis PCI-1149.1/101 card. This is the equivalent, for the purposes of this design guide, of the embedded software in an operational system. This is the program that reads and writes the correct SCANSTA101 STA master registers to perform the built-in self-test. In Chapter 8, the use of ScanVec will be described along with what it looks like when the self-test completes successfully and when it detects a fault.

The designer of a system with boundary scan-based built-in self-test might use ScanVec as a debugging tool in the early stages of development. It must be emphasized, however, that the functions performed by ScanVec in this demonstration system would be performed by an embedded controller in an operational system which implements boundary scan-based self-test.

3.3. Summary

The SCANSTAEVK demonstration kit and the associated software is meant to model, in a simplified way, an operational system implementing boundary scan-based built-in self-test. All of the elements required are present. There is a model of the application board which is to test itself, the target SerDes board. There is a Scan Bridge multiplexer such as might be used to isolate the scan chains in the target application. There is a SCANSTA101 STA master JTAG controller which performs the low-level JTAG TAP port manipulations to carry out the built-in self-test. And there is a system backplane over which the test vectors are delivered, just as there might be in an operational system.

The test patterns are generated by ATPG software just as they would be for an operational system. The test pattern output is converted to EVF2 format just as it would be for an operational system. And, just as in an operational system, a system controller delivers the EVF2 file to the SCANSTA101 STA master over parallel address and data buses by reading and writing the appropriate registers and memory locations.

Producing and deploying test vectors for built-in self-test of this simplified system should provide valuable insight into the process required to do the same thing in an operational system. The following sections of this design guide will describe the process of designing and implementing a built-in self-test and dissect the results in some detail. The intent is to provide a deeper understanding of the boundary scan development and deployment process.

4. Hardware Design Considerations for Built-In Self-Test

The SCANSTAEVK demonstration kit used as a test platform for this work was designed several years ago as a demonstration vehicle for National's family of JTAG support devices. The primary focus of the present effort was to develop and deploy a test vector sequence for use on this hardware in the same way that a system designer would for an operational system. Still, it is instructive to examine some of the considerations involved in the hardware design required for built-in self-test. The SCANSTAEVK demonstration kit will be treated as if it were designed today from scratch.

4.1. System Function and Built-In Self-Test

Built-in self-test is clearly always an adjunct to the primary function of a board or subsystem. So the first step in designing a board which will include built-in self-test is to design the board to perform its primary function. The IEEE 1149.1 boundary scan test standard is focused on providing test capability without compromising the primary function of the board or subsystem under test. Ideally, at the beginning of the design cycle, it should not be necessary for the system designer to consider built-in self-test at all.

National's family of boundary scan support devices was designed with this philosophy in mind. It is almost possible to add boundary scan support at the very end of the schematic design process, just before going to board layout. As a practical matter, of course, some consideration must be given to boundary scan earlier in the design process.

4.1.1. Selection of JTAG-Enabled Devices

It is obvious, though worth stating, that boundary scan is primarily capable of testing interconnects between devices that are equipped with the IEEE 1149.1 TAP. Testing of other interconnects is possible with boundary scan, but the test sequence becomes more complicated. Some ATPG tools can generate automatic test sequences for memory devices and for simple combinatorial logic. But the first requirement in designing a board for built-in self-test is to choose devices, to the extent possible, that are equipped with the IEEE 1149.1 TAP.

National has implemented the IEEE 1149.1 TAP on many devices, and the serializer/deserializer pair on the SCANSTAEVK SerDes target board are a good example. The SCAN921023 serializer and SCAN921224 deserializer are designed for boundary scan testing with input boundary scan cells on all of the CMOS digital inputs and output boundary scan cells on all of the CMOS digital outputs. In addition, these devices have differential boundary scan cells on their primary LVDS inputs and outputs. They also have built-in self-test modes that transmit known data patterns from the serializer to the deserializer to test the primary LVDS link at operational speed. This is an extension to the standard boundary scan tests. If this sort of at-speed test is desired, it must be added explicitly to the system test vectors.

The SCANSTAEVK SerDes target board is an ideal candidate for built-in self-test in this sense because all of the integrated circuits on the board (admittedly, there are only two of them) are equipped with the IEEE 1149.1 TAP.

4.1.2. Selection of JTAG Support Devices

In order to implement board built-in self-test using National's family of JTAG support devices, a few preliminary system-level decisions must be made.

First is the location of the SCANSTA101 STA master. For many applications, locating this device at a single primary location on the backplane (on the system controller board, for example) will work fine. The limitation of this approach is that a single SCANSTA101 STA master can only drive one set of test vectors at a time. Even if a Scan Bridge multiplexer is located on the same card as the SCANSTA101 STA master, providing multiple JTAG scan chains on the backplane, either only one of the scan chains will be active at a time or all the scan chains will be doing the same thing. For systems where it is important that built-in self-test be performed as fast as possible, it may be preferable to locate a SCANSTA101 STA master on each board in the system. Once these devices are set up by the system controller, they can perform the required self-tests autonomously and report the results back to the system controller. This allows testing of the entire system in the fastest possible manner.

Once the decision has been made regarding how many STA masters to use and where to locate them, board space and power from the power supply must be allocated for these devices. This is done in the early stages of the system design.

The SCANSTA101 STA master is designed to require very little additional external logic, but depending upon the system design, some signal conditioning on the parallel processor interface handshake lines might be required. The system designer should have an architecture in mind for connecting the SCANSTA101 STA master(s) to the system controller and for accomplishing this handshaking. If external logic is anticipated, provision should be made for it early in the system design.

The next consideration is the implementation of multiple scan chains using the Scan Bridge multiplexers. If the decision of

how to implement the STA masters has implications for the speed of testing the entire system, the implementation of the Scan Bridge multiplexers has implications for the speed of testing any part of the system independently. A board with many JTAG-enabled devices and many interconnections to be tested might require a very long test time if the devices are connected in a single scan chain. It might be advantageous in this situation to implement multiple scan chains on this board using the SCANSTA111 or SCANSTA112 Scan Bridge multiplexers.

Considerations in the deployment of the Scan Bridge multiplexers include whether there are sections of the board with few operational interconnects between them. If there are sets of devices on the board that perform relatively independent functions in the operational system, these might be good candidates for local scan chains. Additionally, if the JTAG TAP is to be used to program a programmable logic device such as a FPGA, it will speed up the programming process to put the programmable logic device alone on a dedicated local scan chain. This will also simplify the programming of the device.

The SCANSTA111 multiplexer provides three local scan ports, two of which have a one-bit pass-through input and output associated with them. This may be sufficient for many applications. If more local scan ports are required, the SCANSTA112 multiplexer provides seven, of which two have two-bit pass-through inputs and outputs. If more local scan chains than this are required, these devices may be configured in a hierarchical scan chain, with a local port of one multiplexer connected to the master port of the next. Many of the ATPG tools can handle this sort of hierarchical configuration automatically.

Once the number and type of Scan Bridge multiplexers have been determined, board space and power from the power supply must be allocated for these devices as well. This is, again, a decision that should be made early in the system design.

Finally, although this hasn't been discussed yet, if analog voltage monitoring and reporting over the IEEE 1149.1 TAP is desired, National's SCANSTA476 eight-input analog voltage monitor can be implemented on the board. If this device is to be used, board space and power must be allocated for it early in the design cycle as well.

4.1.3. Delivery of the Test Vectors

If the system is to implement built-in self-test, somewhere in it there must be contained a controller with parallel data and address ports to communicate with the SCANSTA101 STA master and sufficient memory to store the on-board test vectors.

In any system complex enough to be considered a candidate for built-in self-test, there is probably already a system controller that meets this requirement and enough memory to store the test vectors.

In the worst case, it might be necessary to add some additional system memory to store the required test vectors. These test vectors are stored in the target system as binary-encoded EVF2 files, which provide for efficient memory use. The system designer should consider early in the design process whether there is sufficient extra memory in the system to store the required EVF2 files, and should plan for sufficient memory to accommodate them.

If there is no controller in the system at all, perhaps the system designer should consider whether built-in self-test is really needed at all, and if so, how it will be initiated and how the results will be reported. A simple system without a system controller is not a good candidate for built-in self-test and probably should be tested in another way.

4.1.4. TAP Connections

The IEEE 1149.1 TAP was designed as a four- or five-line interface specifically to avoid complicating the system interconnection design. Even so, provision must be made for additional backplane connections associated with the built-in self-test function. There are essentially three use models to be considered.

4.1.4.1. Single JTAG TAP over the Backplane

This architecture has been mentioned previously. This is the situation when the SCANSTA101 STA master is located at a single point on the backplane, for example on the system controller board, and when any Scan Bridge multiplexers are located on the boards to be self-tested. In this application, a single set of JTAG TAP lines must be routed over the backplane and through the backplane connectors of each board to be tested. This requires just four or five additional lines on the backplane and through the backplane connectors.

An example of this type of TAP connection is shown in ***Figure 4-1***. The JTAG-enabled devices in this example are distributed among multiple functional boards in the system. The system controller and the SCANSTA101 STA master are located on a system controller board. The JTAG TAP lines, TCK, TMS, and TRST* could be routed through each board or could be tapped off the backplane in parallel for each board. The same TCK, TMS, and TRST* lines go to all the JTAG-enabled devices. The TDI line is an input to each board and the TDO line is an output to the next board in the system shown in ***Figure 4-1***.

The SCANSTA111 or SCANSTA112 multiplexers could be used on any or all boards in the system to partition the scan chain on each board into multiple local scan chains. The single backplane JTAG TAP would then control the SCANSTA111 or SCANSTA112 multiplexers. In such a system, the SCANSTA101 STA master would be responsible for addressing and configuring the SCANSTA111 and SCANSTA112 multiplexers during JTAG operations. National's Application Note AN-1259, SCANSTA112 Designer's Reference[9], describes the usage of the SCANSTA112 multiplexer in detail. The application note also applies to the SCANSTA111 multiplexer.

In essence, this single TAP connection over the backplane is the use model for the SCANSTAEVK demonstration kit. There are actually multiple JTAG TAPs routed over the backplane and through the backplane connectors to the target board, but they are not used in the case study presented in this design guide. Only a single JTAG TAP is used. The SCANSTA101 STA master is on the system controller board which is located in the PC.

4.1.4.2. Multiple JTAG TAPs over the Backplane

This is the case where the SCANSTA101 STA master and one or more Scan Bridge multiplexers are located at a single point on the backplane. In this case, four or five lines for each of the JTAG TAPs are routed over the backplane and the appropriate local scan port or ports are routed through each backplane connector. This arrangement requires more backplane connections than a single TAP would, but it provides some additional flexibility and speed. This configuration may be the best choice for some applications.

An example of this type of TAP connection is shown in ***Figure 4-2***. The JTAG-enabled devices in this example are distributed among multiple functional boards in the system. The system controller, the SCANSTA101 STA master, and, in this example, a SCANSTA112 seven-port multidrop JTAG multiplexer are located on a system controller board. Two local scan ports are shown in ***Figure 4-2***. The TAP lines for each local scan port are routed over the backplane and through the connectors to the functional boards.

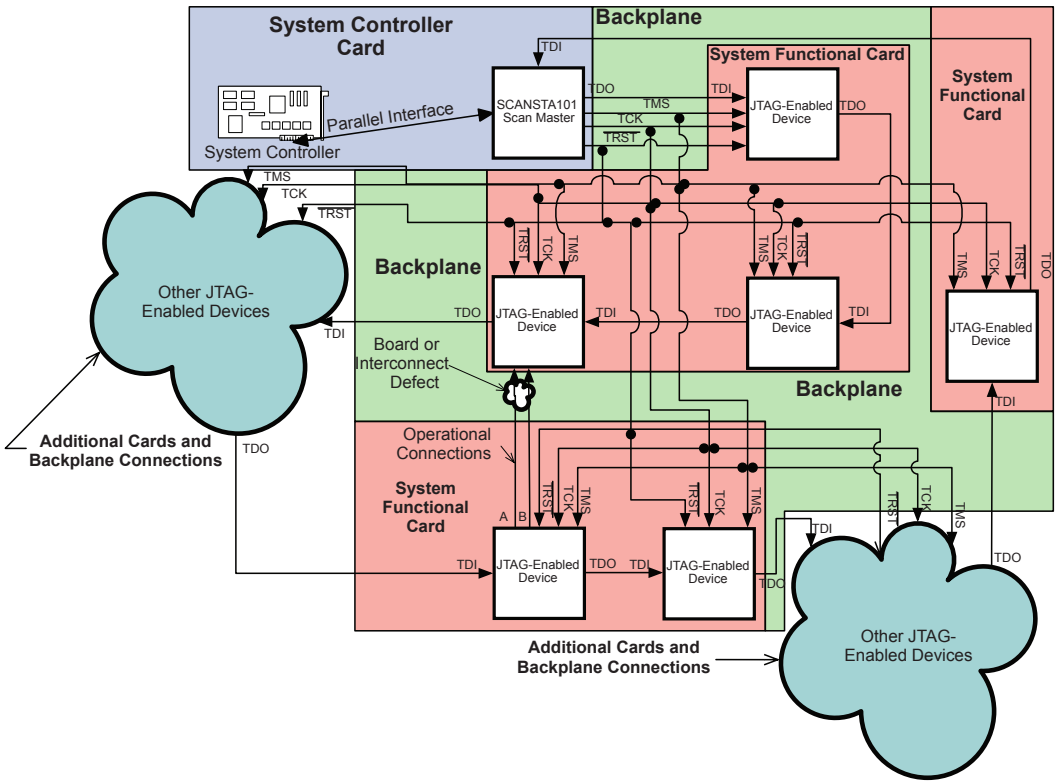

Figure 4-1. Example of a System with a Single JTAG TAP. The JTAG TAP is routed over the Backplane and through the Backplane Connectors.

Where JTAG operations are to be performed that do not apply to all the JTAG-enabled devices in the system, but only to some subset of them, a partitioned scan chain system like that shown in ***Figure 4-2*** can improve the speed of JTAG operations. Only the bits required for JTAG operations on the desired part of the system must be transmitted through the scan chain. Fewer bits imply faster operation. So the speed improvements possible with this sort of system are not realized by shifting bits through the scan chain faster, but by shifting fewer bits at the same speed only through the required part of the scan chain.

4.1.4.3. Parallel-Port Communication over the Backplane

Where each board to be tested is equipped with a SCANSTA101 STA master, the system processor communicates with the STA masters by means of a parallel interface over the backplane and through the backplane connectors. This arrangement can potentially provide the fastest built-in self-test operation. This comes at the cost of potentially more lines on the backplane and through the backplane connectors.

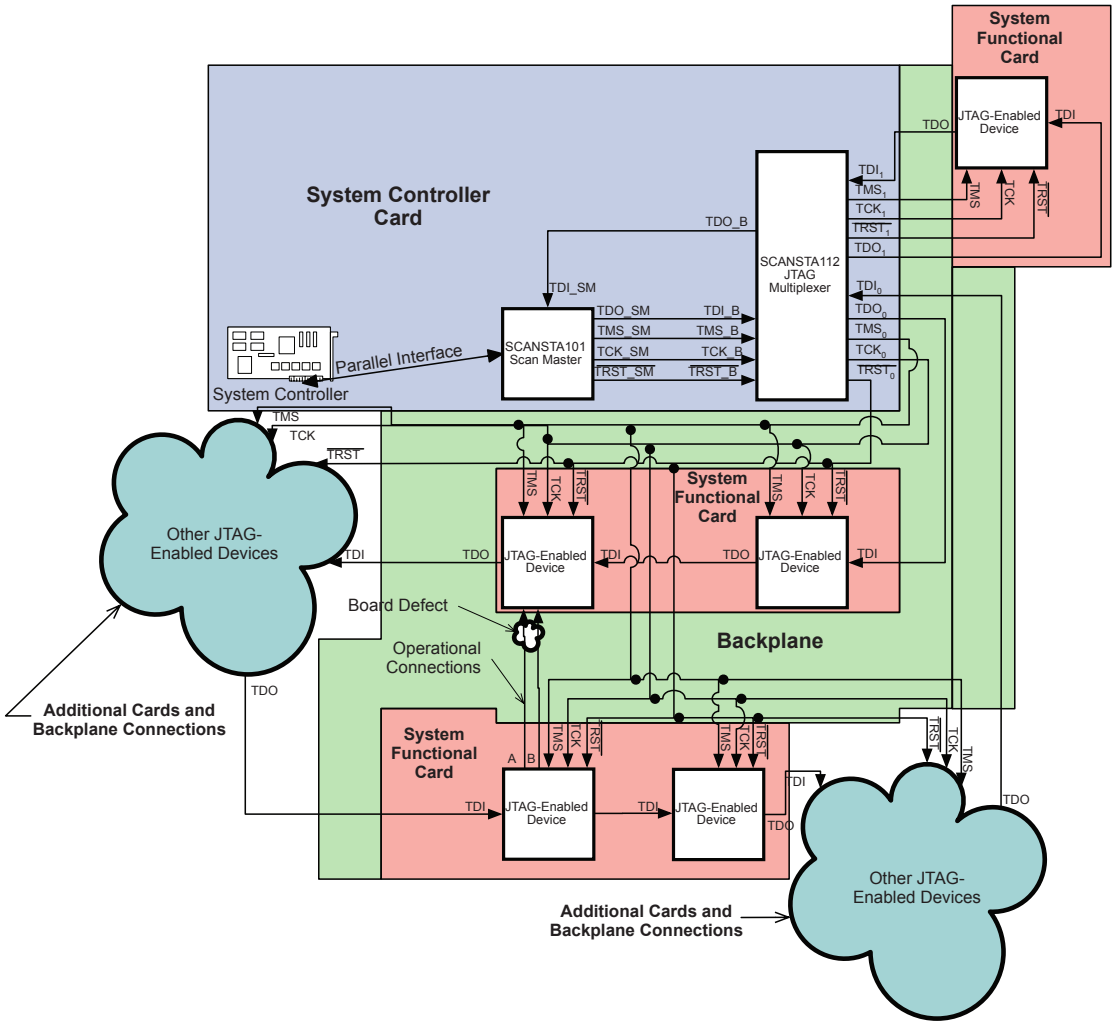

Figure 4-2. Example of a System with Multiple Multiplexed JTAG TAPs. The Local Port JTAG TAPs are routed on the Backplane and through the Connectors.

Hardware Design Considerations for Built-In Self-Test

An example of a system with multiple boards, each equipped with a SCANSTA101 STA master, is shown in ***Figure 4-3***. The parallel interfaces from the system controller to each SCANSTA101 STA master are routed over the backplane and through the connectors. Each scan chain in this system could be partitioned using a SCANSTA111 or SCANSTA112 multiplexer. As is the case in the system with multiple JTAG TAPs, the system shown in ***Figure 4-3*** can improve the speed of JTAG operations by communicating only with the desired subset of JTAG-enabled devices. The scan chain is partitioned by the operation of addressing only the desired SCANSTA101 STA masters and also by addressing only the desired local scan ports of the SCANSTA111 and SCANSTA112 multiplexers.

Obviously combinations of and extensions to these three use models could also be envisioned, but the system-level considerations are the same. However the system designer chooses to implement built-in self-test, there must be enough traces allocated on the backplane and through the backplane connectors to support the additional communication required for the built-in self-test.

Considerations of why and how to partition the JTAG scan chain are beyond the scope of the present document. National publishes an application note discussing partitioning of the JTAG scan chain using the SCANSTA112 multiplexer[9]. JTAG Technologies also publishes brochures on board Design For Test (DFT) guidelines and system DFT guidelines that provide valuable information about scan chain partitioning and signal routing[10] [11].

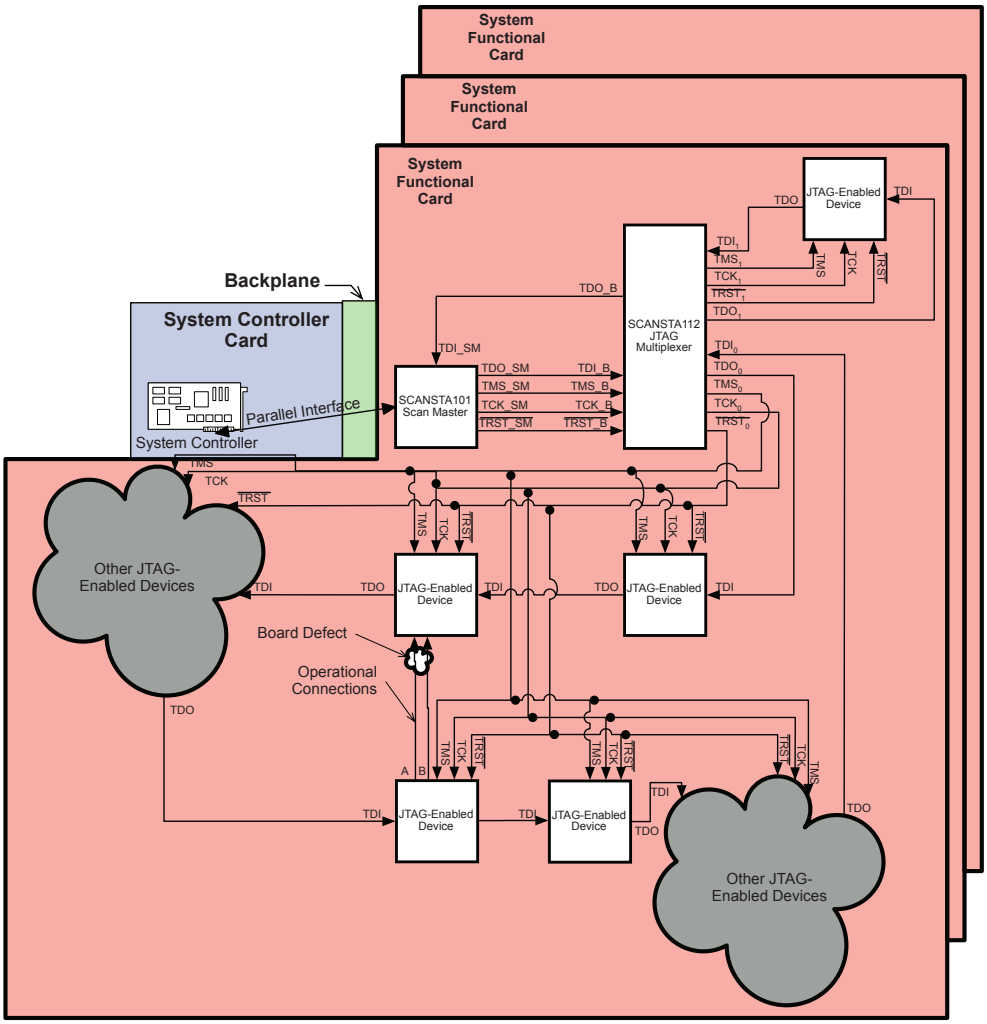

Figure 4-3. Example of a System with Multiple SCANSTA101 STA Masters. The Parallel Interfaces to the SCANSTA101 STA Masters are routed on the Backplane and through the Connectors.

4.2. Test Implementation

Once the previously-described, top-level system considerations have been accounted for, most of the design related to the primary functions of a system can proceed without any further reference to the built-in self-test functionality. The devices in the system can be interconnected in the manner required for them to perform their primary function, whatever it may be. No additional special provisions need to be made for built-in self-test. This is a key feature of the IEEE 1149.1 boundary scan test standard. Its highly-adaptable, light-weight interface makes it simple to "bolt on" boundary scan testing near the end of the design process.

The SerDes target board in the SCANSTAEVK demonstration kit (which is used as the model for the board to be tested in the present effort) was actually designed with boundary scan in mind. That is why it includes provisions for introducing deliberate faults in the interconnections and in the static logic values. These are the faults in the board connections that will be detected using boundary scan. It is easy to see, however, that these specialized provisions for boundary scan testing could be eliminated without compromising the primary functionality of the board. The fact that these provisions are there can provide some insight into the workings of the built-in self-test, but in an operational system these sorts of things (for example, the provision to short DIN4 to ground using S5; ***Figure 3-7***) would not be included.

To reiterate, were the system being designed from scratch, built-in self-test would have been considered only to a limited extent so far. The following would have had to be considered:

1. Selection of devices that implement the IEEE 1149.1 TAP
2. Architecture, location, and power requirements of the JTAG-support devices and allocation of devices to local scan chains
3. Controller and memory requirements
4. Additional connections required on the backplane and through the backplane connectors

Now, as far as the hardware design is concerned, it is time to implement the built-in self-test and only three things are necessary to do so.

4.2.1. Place and Route the JTAG Support Devices

Allocation of power and board space to the JTAG support devices has already been made and a high-level decision has been made as to where they will be placed. The SCANSTA101, SCANSTA111, and SCANSTA112 devices to be used in the system must now be placed on the schematic and connected to the power supplies. All of these devices are highly integrated and self contained, and they require little, if any, "glue logic" to implement them in a system.

Referring to ***Figure 3-5***, this schematic indicates, in the SCANSTAEVK demonstration kit used in this effort, the address of the SCANSTA111 is set by a pair of Binary-Coded Decimal (BCD) switches. In an operational system, this isn't necessary. Something has to set the address of the SCANSTA111 or SCANSTA112 multiplexers, but it does not need to be switches. The address can be set with jumpers or can be hard-wired if desired. The point is that even the modest additional complexity associated with the SCANSTA111 multiplexer in the SCANSTAEVK demonstration kit used for this effort is not necessary in an operational system. Adding the SCANSTA111 or SCANSTA112 multiplexers to a board to implement built-in self-test does not require adding much else to support it. This is inherent in the design of the SCANSTA111 and SCANSTA112 multiplexers.

Likewise, adding the SCANSTA101 STA master to a board in order to implement built-in self-test can be accomplished with very little additional hardware. The block diagram of the SCANSTA101 STA master is shown in ***Figure 4-4***. For a boundary scan-based built-in self-test, all the lines shown on the parallel processor interface block would be connected to the system controller.

4.2.2. Connect the Parallel Processor Interface of the SCANSTA101 STA Master(s)

The only connections to be made to the JTAG-support devices, aside from the JTAG TAPs themselves, are the connections to the parallel processor interface of the SCANSTA101 STA master(s) in the system. These connections consist of a 16-bit-wide data bus, a 5-bit-wide address bus, several handshake lines, a system clock, and reset and output enable lines.

The data bus is simple in concept. The system controller must have some sort of bus for reading from and writing to the onboard memory. The SCANSTA101 data bus could be connected directly to the same bus since, as far as the controller is concerned, all that is required is to read and write 16-bit memory locations. That these memory locations are actually registers in the SCANSTA101 STA master need not affect the process of reading from and writing to these memory locations, though the handshaking required might be different. This can be handled in software in the system controller.

Address translation for the SCANSTA101 STA master might be required. As far as the SCANSTA101 STA master is concerned, its register address space runs from 0x00 to 0x19, but the processor might need to map the address space differently. Since the address bus of the SCANSTA101 STA master is only 5 bits wide, address translation could be as simple as selecting 5 bits from a wider address bus on the system controller. If more sophisticated address translation is required, a small amount of external logic might be added to accomplish this.

Similarly, some digital signal conditioning might be required for the handshake lines between the parallel processor interface and the system controller. This might be accomplished by the controller itself or might require some external logic. The handshaking is designed to be simple so that it can be accomplished by minimal and simple external logic.

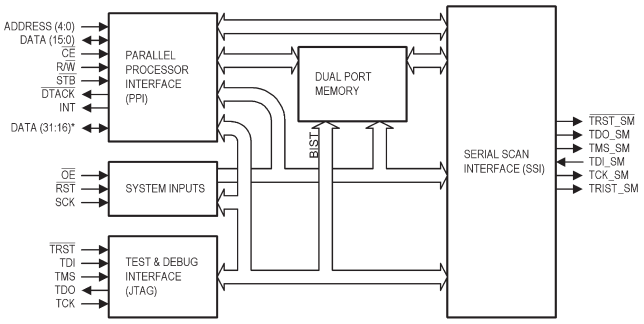

Figure 4-4. Block Diagram of the SCANSTA101 STA Master

The lines shown coming into the system inputs block in the block diagram of ***Figure 4-4*** may also be provided by the system controller, or they may be provided from some other source. The source for these inputs depends upon the system design and the desired method for initiating built-in self-test and for resetting the JTAG TAP. In any case, the amount of external logic required in addition to the SCANSTA101 STA master itself is minimal. This is part of the design philosophy of the SCANSTA101 STA master.

The system clock can be the same as the clock for the system controller or it could be a divided-down version of this clock. The maximum supported system clock frequency is 66 MHz.

4.2.3. Connect the TAP

This is the last step (from a hardware standpoint) in implementing boundary scan-based built-in self-test. In concept, what must be done is to connect the TAP lines from the SCANSTA101 STA master to the TAPs on the JTAG-enabled components on the board to be tested. These lines may be connected directly or they may go through one or more Scan Bridge multiplexers. The connections to the TAPs are independent of the primary functional connections on each device. This is required by the IEEE 1149.1 standard. The handling of the TAP connections is fairly simple and is described in the following sections.

4.2.4. Asynchronous Reset TRST*

Active low signals are designated in this design guide by a trailing asterisk. Other conventions commonly used for this type of signal are an overbar, which is used in some of the figures in this design guide, and a trailing slash.

The asynchronous reset line TRST* is optional in the IEEE 1149.1 boundary scan standard. For devices that do implement this line, it is intended to provide an asynchronous reset of the TAP, sending the TAP state machine back to the Test-Logic-Reset state immediately. Normally this line should be connected from the SCANSTA101 STA master either to every JTAG-enabled device that implements this line or to each of the SCANSTA111 and SCANSTA112 multiplexers, which can pass it through to the local scan ports.

If the multiplexers are used, then the TRST* line of each local scan port should be connected to the TRST* line of each device in its corresponding local scan chain that implements this line.

The TRST* line is meant to fan out, as a single net, to all the devices on a given scan chain. It either does this directly or through a Scan Bridge multiplexer. If the TRST* line is routed through a multiplexer, the local scan port TRST* lines carry a buffered version of the input TRST* line to the multiplexer.

4.2.5. Test Clock (TCK)

The TCK line carries the test clock which clocks the Test Mode Select (TMS), Test Data In (TDI) and Test Data Out (TDO) lines. This signal is meant to fan out, as a single net, to all the JTAG-enabled devices on a board.

When the SCANSTA111 or SCANSTA112 Scan Bridge multiplexers are used, each local scan port has a buffered version of the TCK line on the master TAP. When these multiplexers are used, the TCK output from each local scan port should be connected to the TCK inputs of all the devices on the corresponding local scan chain.

4.2.6. Test Mode Select (TMS)

The TMS line controls the state of the TAP state machine which controls the operation of the TAP. In normal operation of

the IEEE 1149.1 boundary scan chain, the TAP state machines of all the devices on the scan chain are in the same state. To accomplish this, all the TMS lines are connected together on a single net.

When the SCANSTA111 or SCANSTA112 Scan Bridge multiplexers are used, the TMS line from each local scan port should be connected to the TMS inputs of all the devices on the corresponding scan chain. When the multiplexers are used, the TMS lines on each local scan port may not follow the TMS line on the input to the multiplexer. This is by design. It is by controlling the TMS line of each local scan chain that the Scan Bridge multiplexers isolate the operation of each local scan port from the others.

4.2.7. Test Data In (TDI) and Test Data Out (TDO)

Beginning with the SCANSTA101 STA master, the TDO line of each boundary scan device should be connected to the TDI line of the next device in the chain. This connection, in fact, is why this configuration is called a boundary scan chain. When the SCANSTA111 or SCANSTA112 Scan Bridge multiplexers are used, the TDO line of each local scan port should be connected to the TDI line of the first boundary scan device on the corresponding local scan chain, and from there to the other devices on the chain.

In the end, the TDO line of the last device in the chain is connected to the TDI line of the original TDO driver, either the SCANSTA101 STA master itself or the Scan Bridge multiplexer local scan port TDI input. When Scan Bridge multiplexers are used, they form their own scan chain, with the TDO output of the last multiplexer in the chain connected to the TDI input of the STA master.

In boundary scan operation, data is clocked out of the TDO line of the STA master, one bit at a time, into the TDI line of the next device in the chain. The state of the TAP state machine and the internal JTAG logic of this device determine where this data goes, but it eventually is shifted out on the TDO line of this device and into the TDI line of the next one.

This is all that needs to be done to connect the TAP for built-in self-test from a hardware point of view. ***Figure 4-5*** shows an example connection of a JTAG scan chain on a target board using a SCANSTA112 Scan Bridge multiplexer. Each of the LSP connections shown is only five lines wide, and there is no interaction with the basic functionality of the board.

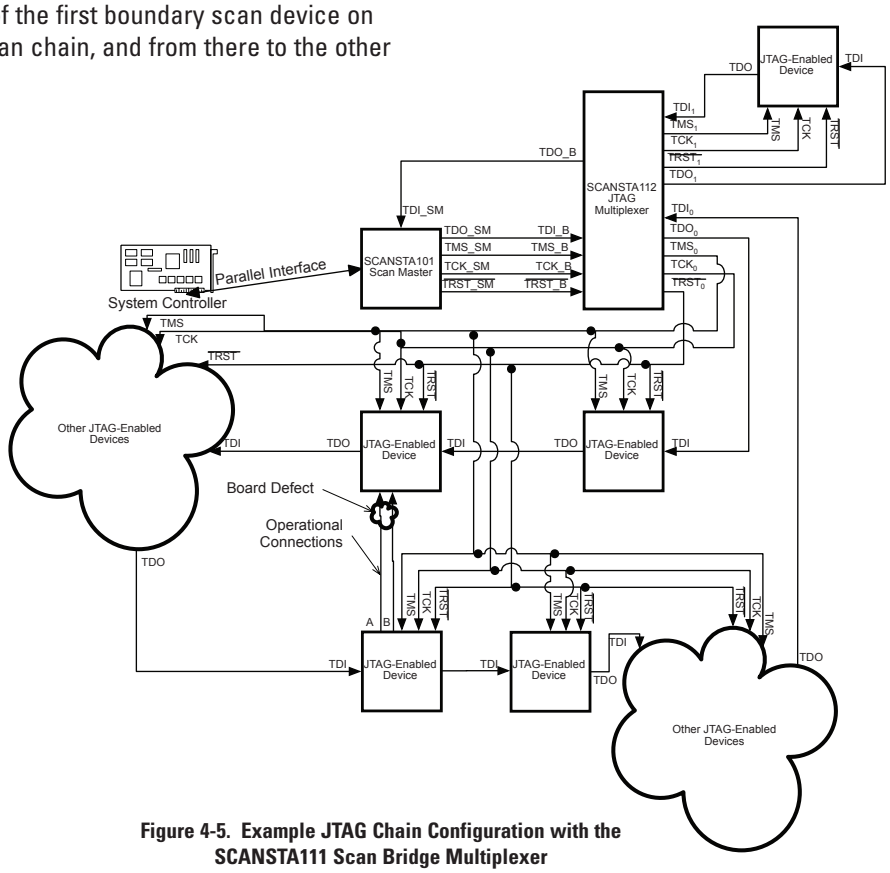

Figure 4-5. Example JTAG Chain Configuration with the SCANSTA111 Scan Bridge Multiplexer

4.3. Summary

As previously mentioned, the hardware design of the SCANSTAEVK demonstration kit used in this effort was done several years ago. The design considerations for this demonstration kit were somewhat different from those for an operational system with built-in self-test functionality included as an add-on. The demonstration kit was designed to demonstrate JTAG functionality. An operational system would just be designed to use it.

The previous sections have described the hardware design considerations for such a system and related them to the design of the SCANSTAEVK demonstration kit. The key points related to hardware design are as follows:

1. The design of the boundary scan-based built-in self-test can largely be deferred until late in the design process. Only some high-level system considerations must be addressed at the beginning of the process.
2. Once the functional design of the system is almost complete, boundary scan-based built-in self-test can be "bolted on" with very little impact to the hardware design of the system.
3. Implementation of the boundary scan chains requires few, if any, new components on the board.
4. Implementation of the boundary scan chains requires few new board interconnects, and none that would disturb the basic functionality of the board.

Thus far, the SCANSTAEVK demonstration kit used as a model of an operational system for the present effort has been described. Also described were some of the considerations in the hardware design of such a system, relating them to the design of the SCANSTAEVK demonstration kit. In the sections that follow, how to develop and deploy a test sequence for the SCANSTAEVK demonstration kit will be illustrated. The first step in this process is to develop a test pattern using commercially-available ATPG software.

5. Development of a Boundary Scan Test Pattern

In the course of this work, ATPG tools from three different tool vendors were evaluated. These tools were Corelis ScanExpressTPG, Flynn Systems OnTAP, and JTAG Technologies ProVision. All three tools were suitable for generating boundary scan-based built-in self-test patterns, and the choice of which tool to use is largely a matter of personal preference and company history. All three of these tools are primarily targeted at producing test patterns to be deployed on hardware provided by the tool vendor. This is, of course, a factory test, and is different from built-in self-test. Having said that, all three tools can also be used to produce SVF files suitable for built-in self-test, and each of the three tools was used in this way to produce such a SVF file.

Other tools are also available for automatic test pattern generation. The fact that only the three tools listed were evaluated for this design guide does not imply any special fitness of these tools for automatic test pattern generation. These were simply the three tools that were evaluated. ATPG tools from other vendors could certainly be used for JTAG-based built-in self-test.

5.1. ATPG Tool Selection

In this design guide, the test preparation sequence using several of the ATPG tools as demonstration vehicles will be described. Again, the choice of an ATPG tool is largely a matter of personal preference. Some of the considerations in the choice of an ATPG tool are outlined in the following list.

1. The system to be tested might be described by a single netlist or by multiple netlists. The SCANSTAEVK demonstration system used is described by multiple netlists. However, since only a single board was fundamentally tested, (even though it was part of a multiple board system, and since the SCANSTA111 Scan Bridge multiplexer being used was really only connected to the TAP), it was easy for the data to be input into any of the tools to produce the desired tests.
2. All the tools produce intermediate text files describing the tests to be performed. It is easy to edit these text files with a standard text editor. As an example, the text files were edited to describe the static logic levels for the desired tests , the connections between the parts not shown on the netlist, and the connection of the SCANSTA111 multiplexer, which was also not on the target board netlist. This method of adding this information by editing text files seemed natural and straightforward, and it was easy to get the necessary behavior by editing these files. That being said, all of the tools are designed so that editing these text files is not necessary. All the required information can be entered into each of the tools using the GUI.
3. Many of the tools support the SCANSTA111 and SCANSTA112 JTAG multiplexers in a straightforward way. However, implementation of a full hierarchical system with multiple Scan Bridge multiplexers was not attempted.
4. Each of the tools can generate a SVF file that can be used for producing an EVF2 file. The SVF files are also ASCII text files that can be viewed or edited with a text editor.

To get started with the development of the boundary scan test, it was first necessary to produce a netlist for the SCANSTAEVK SerDes target board.

5.2. Producing the Netlist

The SCANSTAEVK demonstration kit was designed for National by a third-party design house using Mentor Graphics PADS EDA software. All of the ATPG tools that were evaluated can read a wide variety of netlist formats. These netlist formats include Mentor Graphics PADS, Cadence Allegro, Orcad (which is now part of Cadence), and Altium Designer, the successor to ProTel. Producing a native-format netlist from any of these tools, or from almost any EDA software, is a relatively simple process. So, it was not difficult to produce a netlist for the SCANSTAEVK demonstration kit's SerDes target board. This is the board shown in ***Figure 3-6***, ***Figure 3-7***, and ***Figure 3-8***.

5.3. Entering the Netlist into the ATPG Tool

The following shows a representative section of the SerDes target board netlist.

```
!PADS-POWERPCB-V2007.0-MILS! NETLIST FILE FROM PADS LOGIC
V2007.2
*REMARK* S-03490R0 - SCAN1023-1224.sch -- Thu Oct 01 13:28:53
2009
*REMARK*


*PCB*         GENERAL PARAMETERS OF THE PCB DESIGN

MAXIMUMLAYER 2               Maximum routing layer

*PART*        ITEMS
U1       SCAN921224@BGA\.8MM\49P
U2       SCAN921023@BGA\.8MM\49P
R12      RES\SMT@RES\0603
J1       CONN\DIN\HIR\64P\RA@CONN\DIN\HIR\64P\RA
J10      CONN\3M\2520\20P@CONN\3M\2520\20P
RN1      RSIP8P4R\SMT@RSIP\CTS\744\4R
.
.
.
R16      RES\SMT@RES\2010
M2       FIDUCIAL@FIDUCIAL
*NET*
*SIGNAL* GND
J17.3 J9.2 R1.2 J3.2 C1.2
C6.2 C5.2 C4.2 C3.2 C2.2
J2.11 J10.11 R12.2 RN6.8 RN6.6
RN6.2 RN6.4 RN7.2 RN7.4 RN7.6
RN7.8 J2.16 J10.16 C9.2 R16.1
.
.
.
*SIGNAL* A1_TMS
J13.3 R11.2
*SIGNAL* A1_LSP_ACT
J1.22 R2.1
*SIGNAL* $$$22002
R9.2 J1.15 RN5.4
*SIGNAL* REN
U1.D1 J10.13 RN7.5 S3.8
*SIGNAL* $$$11211
J13.2 U1.G7
*SIGNAL* $$$11228
U1.E5 J12.2
*SIGNAL* $$$11231
U1.E6 J11.2
*SIGNAL* X_\PWRDN
U2.C7 S2.8 RN6.5 J2.20
*SIGNAL* $$$11572
U1.F6 J14.2
.
.
.
```

There are two sections of this netlist that are interesting from the standpoint of automated test pattern generation. First, the section with the heading *PART* includes the reference designators and descriptions for each component on the board. The ATPG tool uses these reference designators when information for each device on the board is entered. For example, as will be seen later, U1 and U2 are JTAG-enabled integrated circuits. Using a BSDL file, information will be provided about their boundary scan implementation and these reference designators will be referred to in order to indicate to which device the BSDL file applies.

Second, the section with the heading *NET*, in which each element is labeled *SIGNAL*, describes the connections between the components on the board. Like most EDA tools, PADS permits but does not require the user to assign each net a unique name. The line labeled *SIGNAL* A1_TMS, for example, and the line after it, indicate that a net labeled A1_TMS (the test mode select line for TAP A1) connects J13, pin 3, to R11, pin 2. This is a net to which the original designer of the board attached a label so that the function of the net would be obvious.

The line labeled *SIGNAL* $$$22002 and the line after it indicate that a net for which the original designer did not enter a name connects R9, pin 2, J1, pin 15, and RN5, pin 4. The software assigned this net a unique name ($$$22002) when the netlist was generated. For the purposes of the ATPG tool, a net name is just a string, and an automatically-assigned net name is as good as any. The use of these automatically-assigned net names in editing the netlist edit file will be described further in Section 5.4.

With reference to ***Figure 3-7***, it should be noted that not all the connections to be tested exist in the netlist or on the schematic. For example, the CAT-5 cable connection between J4 and J5 does not exist in the netlist. In fact, only the connections that correspond to traces on the printed circuit board appear in the netlist. If other connections exist and are to be tested, they must be specified to the ATPG tool in some other way. All of the ATPG tools include provisions for specifying such “off-netlist” connections.

The method of entering the netlist varies from one ATPG tool to another, but entering the netlist is one of the first steps in the process with all of the tools. The main screen of the Flynn Systems onTAP software showing the netlist entry window is shown in ***Figure 5-1***.

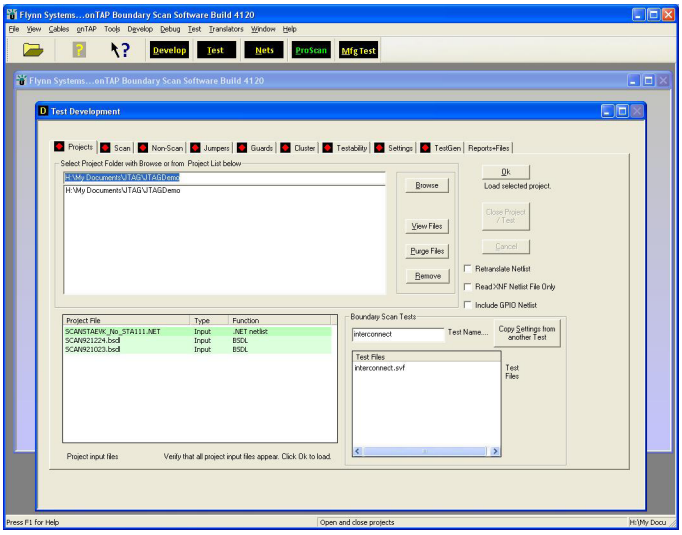

Figure 5-1. Main Screen of the Flynn Systems onTAP Software showing the Netlist Entry Window

5.4. Adding Information to the Netlist

The IEEE 1149.1 standard primarily relates to low-speed or quasi-static testing of interconnections on a board (the newer IEEE 1149.6 standard addresses higher-speed interconnections[12]). Since quasi-static logic levels are applied and sensed in boundary scan testing, some additional information is required to ensure that all interconnects are tested.

5.4.1. Power and Ground Nets

Power and ground nets are the first item of concern. On the SCANSTAEVK SerDes board, none of the boundary scan devices have bidirectional boundary cells (cells which can function either as inputs or outputs). A board that includes devices with bidirectional I/O pins would probably include bidirectional boundary cells. Some of these cells might be connected to a power supply rail, or to ground, to produce static logic levels. If a boundary scan test sequence included an attempt to drive such a bidirectional I/O to some logic level other than the power supply or ground rail it was connected to, it could damage the device. Accordingly, it is necessary to tell the ATPG software about nets that are power supply rails and grounds.

With all the ATPG tools tested, the user designates power and ground nets using the net names from the netlist. This step is prone to error, as it is not always the case that all power and ground nets are labeled with obvious descriptive names. Unfortunately, there is no general way for the ATPG tool to recognize power and ground nets. The user must designate them. One technique for recognizing candidate power and ground nets is to look for nets with a lot of pins connected to them. Such nets are likely to be power or ground nets.

A screen shot of the Corelis ScanExpressTPG main screen is shown in ***Figure 5-2***. The list of net names has been sorted according to the number of pins on each net and the power and the ground net names are obvious. These nets have been designated power and ground as appropriate.

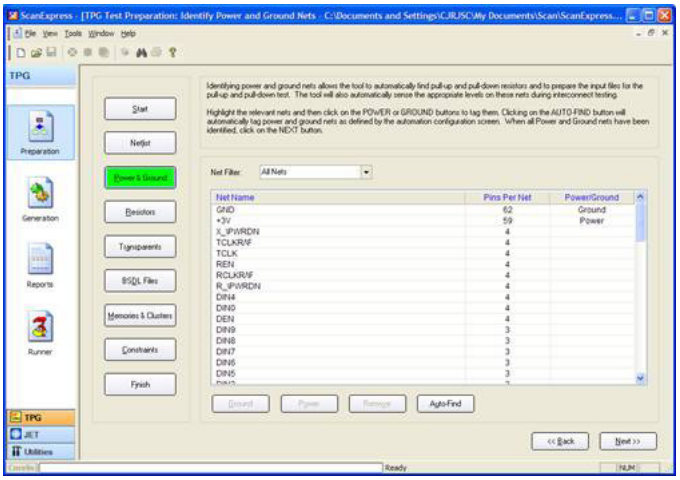

Figure 5-2. Main Screen of the ScanExpressTPG Software showing Power and Ground Net Designation

5.4.2. Resistors

Resistors on a digital board generally serve only a few purposes. They might be pull-up or pull-down resistors used to provide external terminations or to set logic levels on lines that might not be actively driven. These resistors can be considered, for the purposes of testing interconnections with boundary scan, as drivers of "weak zeros" or "weak ones". "'Weak", in this sense, means a logic level that can be overridden by an active driver on the output of an integrated circuit.

Resistors might also be used as transparent signal transmission devices. Zero-ohm resistors are often used in this way on PC boards to support multiple assembly options. Resistors used in this way can generally be treated as wires for purposes of testing interconnections with boundary scan.

Resistors might also be a part of signal conditioning networks such as filters. Since boundary scan testing is implemented with quasi-static logic levels, filter responses are not usually a concern. Such resistors usually can be treated as shorts or opens, depending upon the circuit topology.

Development of a Boundary Scan Test Pattern

All of the ATPG tools include a provision for identifying resistors on the board. Once the resistors are identified, the tools can incorporate the nets that include them into the boundary scan test. The resistors can be designated as pull-ups, pull-downs, or transparent (pass-through) devices.

One other use for resistors, especially in high-speed digital circuitry, is illustrated by R14 on the target SerDes board shown in ***Figure 3-7***. In this figure, the 27Ω resistor is a termination for the high-speed transmission line from the serializer to the deserializer. This resistor is clearly neither a pull-up nor a pull-down resistor, and it must not be treated as a transparent device, either. This resistor has no effect on the boundary scan operations. The ATPG tool can be instructed to ignore this resistor in developing the boundary scan tests.

5.4.3. Transparent Devices

As noted, resistors are in many cases transparent devices (meaning, for resistors, short-circuits) for the purposes of boundary scan testing. There are also other devices that often should be considered transparent, though perhaps not always. These include buffers, switches, multiplexers, and drivers – basically any device that transmits a logic level unmodified from its output to its input. All of the ATPG tools include provisions for identifying transparent devices and include simple models for these devices.

5.4.4. Adding BSDL Models

Arguably the most critical devices on a board to be tested using boundary scan are the JTAG-enabled devices themselves. The IEEE 1149.1 standard specifies extensions to the VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language (VHDL) to describe the boundary scan operation of JTAG-enabled devices. The Boundary Scan Description Language (BSDL) specification uses the generic attributes feature of VHDL to describe the JTAG commands that a device implements, what boundary cells it uses, what bit patterns correspond to each command, which registers each command targets, and other essential information for generating boundary scan test patterns. Following is the BSDL file for the SCAN921023 serializer, shown in its entirety.

```
------------------------------------------------------------
------------------
-- Copyright National Semiconductor Corporation 2001
--
-- Boundary Scan Description Language, BSDL Model for NSC_
SCAN921023
-- 10-bit LVDS Serializer
--
-- National Semiconductor Customer Service Center
-- N. America  (800) 272-9959
-- Europe Germany p49 (0) 69 9508 6208
------------------------------------------------------------
------------------
-- 01    Initial
-- 02    14 Mar 01    Verified through additional ATPG tools
--                     Changed BGA_49 to BGA_49_INTEGER. Added
BGA_49_BALL
--                     Reversed order of DIN from (9 downto 0)
-> (0 to 9)
--                     Corrected ID code
--                     Corrected RUNBIST
-- 03    21 Mar 01    Corrected ID
--                     Corrected cell ordering i.e. cell
closest TDO = 0
-- 04    29 Mar 01    Corrected control cells
-- 05    29 Mar 01    Corrected disable value
-- 06    29 Apr 02    Corrected attribute ordering (RUNBIST_
EXECUTION) & fixed bist register name
-- 07    28 Aug 09    Uncommented DOn and differential port
grouping

entity NSC_SCAN921023 is
     generic (PHYSICAL_PIN_MAP : string := "BGA_49_BALL");

     port (

          DIN:                in          bit_vector(0 to 9);
          SYNC2:              in          bit;
          SYNC1:              in          bit;
          PWRDN:              in          bit;
          DOp:                out         bit;
          DOn:                out         bit;  -- 28 Aug 09
was commented out
          DEN:                in          bit;
          TCLK:               in          bit;
          TCLK_R_F:           in          bit;
          TDI:                in          bit;
          TMS:                in          bit;
          TCK:                in          bit;
          TRST:               in          bit;
          TDO:                out         bit;
          DVCC:               linkage     bit_vector(2 downto
0);
          DGND:               linkage     bit_vector(4 downto
0);
          AVCC:               linkage     bit_vector(4 downto
0);
          AGND:               linkage     bit_vector(4 downto 0)
          );


    use STD_1149_1_1994.all;

     attribute COMPONENT_CONFORMANCE of NSC_SCAN921023 :
entity is "STD_1149_1_1993";



     attribute PIN_MAP of NSC_SCAN921023 : entity is
PHYSICAL_PIN_MAP;

-- BGA_49_INTEGER identifies each pin as an integer
     constant BGA_49_INTEGER : PIN_MAP_STRING :=
          "DIN:(3, 8, 23, 15, 24, 22, 30, 29, 37, 39)," &
          "SYNC2:10," &
```

```
        "SYNC1:4," &
        "PWRDN:21," &
        "DOp:28," &
        "DOn:26," &     -- 28 Aug 09 was commented out
        "DEN:27,"&
        "TCLK:32,"&
        "TCLK_R_F:45,"&
        "TDI:36,"&
        "TMS:31," &
        "TCK:38," &
        "TRST:44," &
        "TDO:43," &
        "DVCC:(17, 18, 33)," &
        "DGND:(1, 16, 34, 40, 46)," &
        "AVCC:(5, 6, 11, 14 ,47)," &
        "AGND:(12, 13, 20, 35, 42)";

-- BGA_49_BALL identifies each pin by a "ball" identifier
    constant BGA_49_BALL : PIN_MAP_STRING :=
        "DIN:(A3,B1, D2, C1, D3, D1, E2, E1, F2, F4)," &
        "SYNC2:B3," &
        "SYNC1:A4," &
        "PWRDN:C7," &
        "DOp:D7," &
        "DOn:D5," &    -- 28 Aug 09 was commented out
        "DEN:D6,"&
        "TCLK:E4,"&
        "TCLK_R_F:G3,"&
        "TDI:F1,"&
        "TMS:E3," &
        "TCK:F3," &
        "TRST:G2," &
        "TDO:G1," &
        "DVCC:(C3, C4, E5)," &
        "DGND:(A1, C2, E6, F5, G4)," &
        "AVCC:(A5, A6, B4, B7 ,G5)," &
        "AGND:(B5, B6, C6, E7, F7)";

    attribute PORT_GROUPING of NSC_SCAN921023 : entity is
-- 28 Aug 09 was commented out
        "DIFFERENTIAL_VOLTAGE ( (DOp, DOn))";
-- 28 Aug 09 was commented out

    attribute TAP_SCAN_IN of TDI : signal is true;
    attribute TAP_SCAN_MODE of TMS : signal is true;
    attribute TAP_SCAN_OUT of TDO : signal is true;

    attribute TAP_SCAN_CLOCK of TCK : signal is (25.0e6,
BOTH);
    attribute TAP_SCAN_RESET of TRST : signal is true;

    attribute INSTRUCTION_LENGTH of NSC_SCAN921023 : entity
is 8;

    attribute INSTRUCTION_OPCODE of NSC_SCAN921023 : entity
is
        "BYPASS           (11111111)," &
        "EXTEST           (00000000)," &
        "SAMPLE           (10000010)," &
        "IDCODE           (10000001)," &
        "CLAMP            (10000111)," &
        "HIGHZ            (00000110)," &
        "RUNBIST          (10000011)";

    attribute INSTRUCTION_CAPTURE of NSC_SCAN921023 : entity
is "XXXXXX01";

    attribute IDCODE_REGISTER of NSC_SCAN921023 : entity is
        "1000" &                 -- version
        "1111110000100110" &    -- part number FC26 TX
        "00000001111" &          -- manufacturer's identity
        "1";                      -- required by 1149.1

    attribute REGISTER_ACCESS of NSC_SCAN921023 : entity is
        "BYPASS           (BYPASS, CLAMP, HIGHZ)," &
        "BOUNDARY         (SAMPLE, EXTEST)," &
        "BISTREG[2]       (RUNBIST)," &
        "DEVICE_ID        (IDCODE)";

--    attribute BOUNDARY_CELLS of NSC_SCAN921023 :entity is
"BC_1,BC_4";

       attribute BOUNDARY_LENGTH of NSC_SCAN921023 : entity
is 18;

    attribute BOUNDARY_REGISTER  of NSC_SCAN921023 : entity
is
     --
     -- num   cell   port                    function   safe
[ccell disval rslt]
     --
        "17  (BC_4,  DIN(8),            input, X)," &
        "16  (BC_4,  DIN(7),            input, X)," &
        "15  (BC_4,  DIN(6),            input, X)," &
        "14  (BC_4,  DIN(5),            input, X)," &
        "13  (BC_4,  DIN(4),            input, X)," &
        "12  (BC_4,  DIN(3),            input, X)," &
        "11  (BC_4,  DIN(2),            input, X)," &
        "10  (BC_4,  DIN(1),            input, X)," &
        "9   (BC_4,  DIN(0),            input, X)," &
        "8   (BC_4,  SYNC2,             input, X)," &
        "7   (BC_4,  SYNC1,             input, X)," &
        "6   (BC_4,  PWRDN,             input, X)," &
        "5   (BC_1,  DOp,               output3,       X,
4, 0, Z)," &
        "4   (BC_1,  *,                 controlr,      0),"
&
        "3   (BC_4,  DEN,               input, X)," &
        "2   (BC_4,  TCLK,              input, X)," &
        "1   (BC_4,  TCLK_R_F,          input, X)," &
        "0   (BC_4,  DIN(9),            input, X)";

   attribute RUNBIST_EXECUTION of NSC_SCAN921023 : entity is
      "Wait_Duration (10.0e-3), "&
      "Observing HIGHZ At_Pins, "&
      "Expect_Data 01";

end NSC_SCAN921023;
```

It should be noted that this is a BSDL file, but it is also a VHDL file. VHDL is a commonly-used hardware description language[13 14]. Most of the information in the BSDL file is not used in VHDL simulations or in the VHDL synthesis process. VHDL permits the addition of user-defined attributes, and this was the mechanism chosen in the IEEE 1149.1 standard to describe the boundary scan operation of a device in a BSDL file. This BSDL file specifies a number of items.

First, it is a VHDL entity description with a generic constant parameter, PHYSICAL_PIN_MAP. Like any VHDL entity description, it describes the interface of the device. The entity declaration specifies the ports of the device, and everything else in the file is an entity declaration item.

Development of a Boundary Scan Test Pattern

The BSDL file specifies the following information used by the ATPG tools:

- Use of the IEEE 1149.1 package STD_1149_1_1994
- The mapping of device ports to physical pins of the device as referred to on the netlist
- The differential pair that represents the LVDS output
- The ports associated with the TAP, which the ATPG tool needs to know about in order to generate the test pattern
- The instructions the device will support
- The corresponding op-codes for each instruction
- Which register each instruction targets
- The expected output bit patterns for the Capture-IR state and for the IDCODE instruction
- Which standard boundary register cells are used and how they are arranged in the boundary register, including the mapping from the boundary register to the ports of the device.

All of the ATPG tools require specification of the BSDL files for each JTAG-enabled device to be tested. Some of the tools provide some BSDL files, but these are mostly for examples. The best way to obtain a BSDL file for a device is to get it from the manufacturer of the device. National provides BSDL files for all of its JTAG-enabled devices.

5.4.5. Modifying the Connections in the Netlist

So far all the additional information added to the netlist has been simply to clarify the information contained in the netlist itself. For example, which nets are power and ground and how various devices should be treated in generating the boundary scan tests have been specified. Some of the information required to develop a boundary scan test is not contained in the netlist, however, and so this information must be provided to the ATPG tools in some other way.

A PC board netlist generally contains only information that relates to the connections made by traces on the board. When a board is deployed in a system, however, it usually includes other connections that are not shown on the netlist. Examples include jumper connections used to set operational conditions, cables between the various boards, switch settings, and interconnections between multiple boards in the same scan chain system.

One way to include all of this information in the system description used to generate the boundary scan tests is to modify the netlist itself. This can be done with a text editor and is straightforward, although it is prone to error. All of the ATPG tools provide techniques to modify the information in the netlist without modifying the netlist itself.

All of the ATPG tools, for example, provide a means for adding connections that are not shown on the netlist. Once these connections have been specified, they can be tested as part of the boundary scan test.

Another example of information that might be added to a netlist is information about static logic levels. All of the ATPG tools allow the specification of static logic levels to be tested. The procedure used to specify static logic levels in the JTAG Technologies tool, for example, is shown in ***Figure 5-3***. The shaded items reading "Sense 1" and "Sense 0" were manually added, and they indicate to the tool that the test pattern should include detection of those static logic levels on those nets.

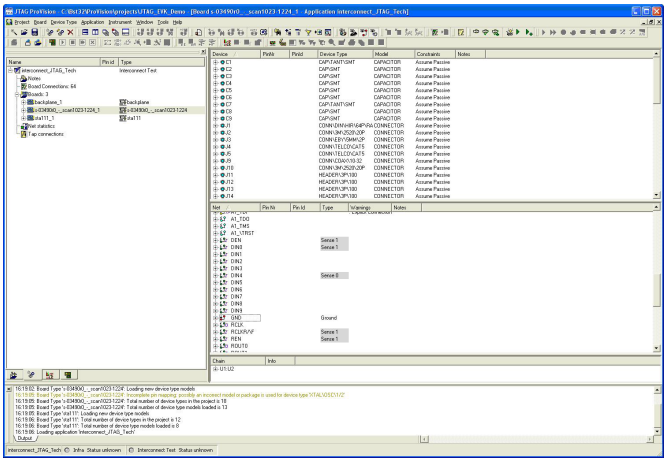

Figure 5-3. JTAG Technologies ProVision Netlist Explorer View showing Specification of Static Logic Levels for Testing

For all of the ATPG tools, the process of generating a boundary scan test specification proceeds in several steps. First, information about the board to be tested, the components on the board, and any additional information not on the netlist are input into the tool. Then the algorithms for test pattern generation are invoked and the boundary scan test specification is produced.

In order to generate and deploy a built-in self-test using National's SCANSTA101 STA master, with or without any additional boundary scan support devices, the boundary scan test specification must eventually be converted to a sequence of register reads and writes that instruct the SCANSTA101 STA master how to perform the required tests. This conversion is

performed by a software tool, EVF Workbench, supplied by National. EVF Workbench requires as input a test specification in Serial Vector Format (SVF). The following section discusses the generation of the boundary scan test and its presentation as a SVF file.

5.5. Generation of the Boundary Scan Test and SVF File

Once the netlist information and any additional information have been supplied to any of the ATPG tools, the next step is to generate one or more sets of test vectors. Considerable effort has been applied to the problem of generating useful sets of test vectors[15] [16] [17]. Each tool generates the test vectors differently, but each tool is designed to produce a set of test vectors that will detect as many board faults as possible with as few test vectors as possible.

5.5.1. Boundary Scan Test Strategy

Fundamentally, what the boundary scan test does is drive known test data on the accessible outputs of the JTAG-enabled devices and look for the corresponding known test data at the inputs of the JTAG-enabled devices. In principle this allows a boundary scan built-in self-test sequence to detect broken or bridged connections on or between the JTAG-enabled devices on the board.

Consider, however, the case where an input line to a JTAG-enabled device is shorted on the board to the positive power supply. No matter what values the driving devices on the line attempt to drive, the JTAG-enabled device at the receiving end of the line will see logic high. If the boundary scan built-in self-test sequence is such that the driving devices on the line only try to drive a logic high, then the board fault will not be detected. The test sequence will only look for logic high, will only see a logic high, and will not detect the fault.

In order to be sure that these sorts of faults are detected, it is necessary to drive multiple logic values on each of the accessible driver lines. If the lines are wire-ORed together, however, and each of the drivers on the line can be driven independently, the important question becomes: How many different values must be driven to ensure detection of all faults that may exist on the board?

It is also possible that logic lines might be shorted to adjacent logic lines and not just to a power supply rail. Then, it is important to determine how many different drive values are required in order to detect such a fault.

The more logic values must be driven and detected, the longer the self-test will take. Therefore, there is a penalty for excessively long test sequences just as there is a penalty for test sequences that are too short to detect existing faults on the board. Automatic generation of a board test pattern utilizes algorithms, unique to each tool, to generate the shortest possible set of tests that will detect all the possible faults on the board. The literature describes the basic strategies for generating these test vectors, but each tool vendor implements these strategies differently. As a result, the test sequences generated by each tool are different. They have different numbers of unique test vectors, and the test vectors are of different lengths.

On a complex board, the number and length of the test vectors required to produce an acceptably-complete board test might be very large, and the corresponding test time might be very long. For the simplified model system used in the present effort, all the tools produced test sequences of comparable length, and all were short enough so that test time was not an important factor. Test time should be considered in a complex system, however, and the efficiency of the test sequence generated by one tool versus another might be an important consideration in the design of a board built-in test sequence.

5.5.2. Serial Vector Format

Serial Vector Format (SVF) is a generic ASCII file format for specifying a sequence of boundary scan-related instructions. For a complete description of the SVF format, see the document Serial Vector Format Specification, maintained by Asset InterTech, Inc[18].

At this point, just to give the reader a flavor of what a SVF format file contains, a section of the SVF file from the Corelis ScanExpressTPG tool has been reproduced.

```
!-------------------------------------------------------------
----------------
!
!            NOTE: Text comments are marked with a preceding
'!' character
!            NOTE: MASK bit value of '0' masks out the
relevant TDO bit
!            NOTE: Serial-data-out (TDI) is shifted out with
LSB first
!
!-------------------------------------------------------------
----------------

TRST OFF;

STATE IDLE;
!-------------------------------------------------------------
----------------
!
```

Development of a Boundary Scan Test Pattern

```
! Source CVF file generated by ScanPlus TPG Version 2.03
!
!-------------------------------------------------------------
---------------
SIR         8  TDI (09)
               TDO (00)
              MASK (00);

SIR         8  TDI (8E)
               TDO (25)
              MASK (FF);

SDR         8  TDI (01)
               TDO (00)
              MASK (00);

SIR         8  TDI (E7)
               TDO (25)
              MASK (FF);

!-------------------------------------------------------------
---------------
!
! TOPOLOGY INFORMATION
! The devices are listed in the order from TDI to TDO of the
board
!
!   STA_U1       (SCANSTA111 Enhanced Scan Bridge at address
0x00000009 on TAP0)
!                          - Instruction Length ( 8)
Boundary Length (  1)
!   U1           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 19)
!   U2           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 18)
!   PAD          (on TAP0) - Instruction Length ( 1) Boundary
Length (  1)
!
!-------------------------------------------------------------
---------------

SIR        25  TDI (1FF0505)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (6DFAAFFFFF)
               TDO (0000000000)
              MASK (0000000000);

SIR        25  TDI (1FE0001)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (4EFB3FFFFF)
               TDO (00E807FC96)
              MASK (00E807FF96);

SDR        39  TDI (77FC3FFFFF)
               TDO (00E807FE96)
              MASK (00E807FF96);

SDR        39  TDI (47FFCFFFFF)
               TDO (00E807FE96)
              MASK (00E807FF96);
```

The first real SVF statement in this file is the TRST OFF statement. This is an instruction to disable the use of the asynchronous TRST* line. It instructs any process that uses the instructions in this SVF file that the asynchronous TRST* line is not to be used. The OFF specification means that this line is to be held inactive during the processing of the SVF file.

The next statement is a STATE IDLE statement. When the keyword following the STATE instruction is one of four stable TAP states, like IDLE, then the instruction specifies that the TAP is to be driven to the indicated state by a correct sequence of TMS line values. After the STATE instruction is executed, the TAP will be left in the Run-Test-Idle state for the next command.

Following is a fragment of the SVF file generated by the JTAG Technologies ProVision tool.

```
! SVF File: "c:\Bst32\ProVision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.svf"
! Input GEN File: "c:\bst32\provision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.gen"
! Input APL File: "c:\Bst32\ProVision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.apl"
! Input CON File: "c:\bst32\provision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.con"

FREQUENCY 25000000 Hz;

STATE RESET;
SIR 8
 TDI  (09);
SIR 8
 TDI  (8e);
SDR 8
 TDI  (01);
SIR 8
 TDI  (e7);
!
! Load Sample/Preload Instruction
!
SIR 25
 TDI  (1550504);
SDR 70
 TDI  (000000003f07f80060);
!
! Load Test Instruction
!
SIR 25
 TDI  (1540000);
SDR 70
 TDI  (000000000400080020)
 TDO  (03f03c07c0e8000694)
 MASK (03ffffffc0e8004694);
SDR 70
 TDI  (000000001c00080020)
 TDO  (03f03c07c0c8000494)
 MASK (03ffffffc0e8004694);
```

This file also includes comments at the top, prefaced by an exclamation point ("!"). The first real SVF instruction in this file is a FREQUENCY instruction. This instruction indicates to the test vector deployment system (the system processor) that the Test Clock (TCK) frequency should not exceed 25 MHz. This specification is contained in the Boundary Scan Description Language (BSDL) files for the devices in the system.

The next instruction in this SVF file is also a STATE instruction. In this case, the TAP is commanded to the Test-Logic-Reset state. The TAP controller will be sequenced through the set of TAP controller states required to perform the specified data shift operations starting from the Test-Logic-Reset state.

The next command in both files is a Scan Instruction Register (SIR) command. This particular command is an address command to the SCANSTA111 multiplexer, which has its 8-bit address statically set to 09. The commands sent to the various devices will be discussed in more detail later.

The other command in these fragments of SVF files is the Scan Data Register (SDR) command. Both the SIR and SDR command include bit sequences to be shifted into the TDI input of the first device in the scan chain. Bit sequences expected at the TDO output of the last device in the chain may also be included. Physically, in a partitioned scan chain like this one, the first and last devices in the chain are the same device, the SCANSTA111 multiplexer in this case. The difference between the two commands is that when a SIR command is executed, the TAP is controlled so that its state machine is sent to the Shift-IR state, whereas when the SDR command is executed, the TAP state machine is sent to the Shift-DR state.

Following is one last example, the SVF file produced by the Flynn Systems onTap tool.

```
// Created by Flynn Systems onTAP Boundary Scan Software
Build 4120,  Fri Sep 11 10:39:29 2009

! DeviceTypes: U1_A2=SCAN921224 U2_A2=SCAN921023
! onTAP Option Selections:
!  (ICT)Single Device Tests Option.............NOT
SELECTED
!  Allow Self Capture(monitoring) on BIDIR Pins...SELECTED
!  Test All Bus Pins(BUS-WIRE Test).............SELECTED
!  Test PULL-UP and PULL-DOWN Resistors..........SELECTED
!  (ICT)Do Not Use Tester Probes................NOT
SELECTED

! onTAP Test and Chain Selections:
!  Chain    Device                 Type
Test
!  A        U1_A2                  SCAN921224
Interconnect
!  A        U2_A2                  SCAN921023
Interconnect

! SVF PIO Characters: H = Drive Logic 1,  L = Drive Logic 0.
!                     U = Detect Logic 1, D = Detect Logic
0.
!                     Z = Drive High Impedance, X = Detect
Unknown.
!PIOMAP  Map package pins to pin names.
!        ATE drives and senses these pins.
! Begin Test Program
ENDIR IDLE;
ENDDR IDLE;
HDR    0;
HIR    0;
TDR    0;
TIR    0;

! Begin BSD cell and port map for U1_A2, SCAN921224
! Pin count = 49
! Pin connections and board level I/O are shown.
!  num cell   port     pin function safe ccell disv rslt I/O
joins signal              probe
!
!    0 BC_1  *             controlr    0
```

The first two instructions in this file are ENDIR IDLE and ENDDR IDLE. The first instruction specifies that at the end of each SIR instruction, the TAP controller is to be left in the Run-Test/Idle state. The second instruction says the same thing for SDR instructions.

The SVF file specifies what the test sequence is to do. That is, it may specify that the TRST* line is, or is not, to be used in the test sequence. It may specify a maximum test clock frequency. It specifies the state from which the test sequence is to begin. It may specify the ending TAP state for the various types of instructions. And it specifies what data is to be shifted into the instruction register of each device through the TAP and what data is to be shifted into and out of the data register in order to perform the test.

Some set of software and hardware must be used to execute the test sequence. This is what the SCANSTA101 STA master is designed to do.

The SCANSTA101 STA master provides a simplified parallel port interface between a system processor and the serial TAP. Once a test sequence has been specified by a SVF file, the system processor must be programmed to convert the instructions in the SVF file into register reads and writes to the SCANSTA101 STA master which will perform the specified test. Fortunately, this conversion can be done in advance without an additional burden on the system processor. A PC-based program called EVF Workbench performs this conversion, and it will be described in the following section.

5.6. Generation of an EVF2 File using EVF Workbench

The simplest way to use the SCANSTA101 STA master to enable a built-in self-test is the multi-step process described in this design guide; namely, generate the self-test description in a SVF file using an ATPG tool, convert it to an EVF2 file using National's software, and deliver it to the SCANSTA101 STA master using National source code functions. The EVF2 format is a binary format that makes efficient use of memory in an embedded processor system. It describes the values to be written to and

read from the various registers in the SCANSTA101 STA master in order to implement the self-test.

For several years, National has provided a command-line-driven utility called svf2evf for converting SVF files to the EVF2 format. This utility has been extensively tested and is stable. It is also fairly easy to use, relying for most settings upon a configuration file that seldom requires modification.

National has recently developed a graphical user interface tool for the SVF to EVF2 format conversion called EVF Workbench. At present, this tool is only available for Microsoft Windows. For Windows users, a graphical tool provides a much more uniform and intuitive interface, and it makes the conversion both simpler and less prone to error. The underlying conversion software is the command-line-based svf2evf program. EVF Workbench can produce a script file that can be run from the command line or can run the script from within the graphical user interface. A screen shot of the front panel of EVF Workbench is shown in ***Figure 3-10***.

Conversion of the SVF file to EVF2 format is simple and fast. The user specifies a configuration file, an input SVF file, and an output EVF2 file. The configuration file describes the parameters of the target device, in this case the SCANSTA101 STA master. The input SVF file describes the test to be performed. The output EVF2 file contains the binary-formatted instructions to be executed by the system controller when the self-test is run.

EVF Workbench also provides the user with a panel of intuitive controls for setting the arguments passed to svf2evf. These parameters would be input on the command line when svf2evf is run from the command line. Having these arguments set from graphical controls makes it easier for the user to specify the parameters for the conversion process.

The process of converting the SVF file to EVF2 format can be performed from EVF Workbench, or it can be run from the command line using the EVF Workbench script output (.ecs) file. The conversion is fast and, if run from inside EVF Workbench, produces diagnostic outputs that are displayed in the text display section at the bottom of the main EVF Workbench window.

EVF Workbench provides a simple interface for conversion of the SVF file to an EVF2 format. Once this conversion has been completed, the EVF2 file contains the information required to run the built-in self-test. Delivery of this information by the system processor to the SCANSTA101 STA master is the subject of the next section of this design guide.

6. Embedded Vector Delivery Software

The steps taken so far to produce a board capable of built-in self-test using the IEEE 1149.1 TAP are as follows:

1. Design the board for its primary function, choosing JTAG-enabled devices where possible and allocating board space and power supply capacity for the additional JTAG support devices that will be required to implement the built-in self-test
2. Design in the JTAG support devices and connect them to the system processor and to the TAPs of the JTAG-enabled devices
3. Generate a test pattern using a commercially-available ATPG tool
4. Convert the test pattern into National's EVF2 format

The EVF2 format specifies the sequence of register reads and writes to the SCANSTA101 STA master to perform the built-in self-test. The next step is to program the system controller to perform this sequence of register reads and writes. National provides software to be embedded into the system controller to assist in delivering the vectors to the SCANSTA101 STA master.

6.1. Development of the System Controller Software

Any system that includes boards to be tested by a built-in self-test must also include some provision for initiating the built-in self-test and reporting the results. Often this function is performed by an embedded controller in the system. The embedded controller must be programmed in some programming language such as C or C++. Even if another language is chosen for the embedded controller, compiled C or C++ libraries often can be linked in to the final program.

To support the delivery of EVF2 format test vectors to the SCANSTA101 STA master, National provides source code which can be integrated into the program for an embedded system controller. This software is designed to provide an interface between the EVF2 format file stored in the system's memory and the low-level register reading and writing functions designed into the system controller.

In order to use the National source code, the system software designer must provide some low-level support facilities. These are described in this section. In this section it is assumed that the system controller is programmed in C. If the system controller is programmed in a different high-level language, or in assembler, it will be necessary for the system software designer to control the interface to the National-supplied source code functions. This is beyond the scope of the present design guide.

6.1.1. National Semiconductor Code

From the perspective of the system software designer, National's source code is simple and modular. Three new files must be compiled and linked to the existing embedded system controller code. These files are as follows:

- EVF2Delivery.h – This header file contains definitions and function prototypes required for the EVF2Delivery functions.
- ST101Def.h – This header file contains the register addresses for the SCANSTA101 STA master and some macros required for the EVF2Delivery functions.
- EVF2Delivery.c – This file may also have a .cpp extension, but it is maintained in standard ANSI C. This file contains the functions required for delivery of the EVF2 vectors.

The EVF2 vector delivery process, from the point of view of the system programmer, consists of providing interfaces to the EVF2 file on the embedded system, providing interfaces to write and read registers in the SCANSTA101 STA master, and using the function EVF2VectorDelivery to connect these two interfaces. Following is the prototype for the function EVF2VectorDelivery.

```
int EVF2VectorDelivery(int (*pfGetData)(void *,void *,void
*,size_t,size_t),
int (*pfErrorHndlr)(const char *,void *,void *),
int (*pfFailHndlr)(unsigned long,void *,void *),
void (*pfRegTraceHndlr)(void *,void *,unsigned
          short,unsigned long,unsigned long),
void (*pfDebugHndlr)(void *,void *,unsigned
          long,unsigned long,unsigned long,
    unsigned long,unsigned long),
void (*pfPollingHndlr)(void *,void *,unsigned long),
unsigned long  dwTimeoutMs,
unsigned long  dwLatencyMs,
void *pvUser1,void *pvUser2);
```

The arguments to this function specify the interface to the EVF2 file and to the error-detection and handling mechanisms. These are described in a later section.

Also included in the EVF2Delivery.c file are prototypes and definitions for the functions used to read and write the registers in the SCANSTA101 STA master. Following is this section of the file.

```
/*
** Prototypes for the specific i/o functions that are currently
**   implemented in another module and resolved during final linkage.
** These could be replaced or removed depending upon the transport
** mechanisim used to access the STA101.
*/
void Write16BitRegister(unsigned short wAddr,unsigned short wData);
void Read16BitRegister(unsigned short wAddr,unsigned short *pwData);
void Write32BitRegister(unsigned short wAddr,unsigned long dwData);
void Read32BitRegister(unsigned short wAddr,unsigned long *pdwData);

/*
** These macros are now defined for the a specific hardware based
** implementation. Simply define these macros to use the appropriate
** i/o functions for the hardware in use.
*/
#define READ16REG(addr,wval)   Read16BitRegister((addr),&(wval))
#define WRITE16REG(addr,wval) Write16BitRegister((addr),(wval))
#define READ32REG(addr,lval)   Read32BitRegister((addr),&(lval))
#define WRITE32REG(addr,lval) Write32BitRegister((addr),(lval))
```

The macros READ16REG, WRITE16REG, and so forth are used in the EVF2VectorDelivery function to read and write the registers in the SCANSTA101 STA master. Functions with these names and argument lists must be supplied by the user. These are described in a later section of this design guide.

Once the appropriate interface functions are available, all the system controller has to do to perform a self-test as specified by the EVF2 file is call the function EVF2VectorDelivery. This function then handles reading the data in from the EVF2 file, interpreting its content, and manipulating the registers of the SCANSTA101 STA master to perform the test. The source code for this function and for the ancillary functions it uses is supplied in the file EVF2Delivery.c so that the user can customize it if desired, but the intent is that this function is platform-independent and can run properly on any system controller equipped with an ANSI C compiler.

6.1.2. Interface to the EVF2 File and Error Handling

In concept, what the EVF2VectorDelivery function does can be described in three steps. First, it reads the EVF2 file from system memory or wherever it is stored. Then it interprets each record in the EVF2 file and determines the required register reads and writes associated with that record. Finally it performs the desired register read and write operations on the SCANSTA101 STA master.

The user is required to provide the EVF2VectorDelivery function with a method for reading data from the EVF2 file and handling errors. This is encapsulated in the argument list to EVF2VectorDelivery.

6.1.2.1. EVF2 File Read Function

The first argument to EVF2VectorDelivery is a pointer to a function returning an int and taking 5 arguments. An example of this function, used in the SCANSTAEVK demonstration kit described in this design guide, follows:

```
int FileCallback(void *pUser1,void *pUser2,void *pData, size_t nSize,size_t nCount);
```

In the implementation used in the SCANSTAEVK demonstration kit software, the first argument, pUser1, is a pointer to the EVF2 file, which is stored on disk in the PC. The second argument, pUser2, is a pointer to the user interface main window. The third argument, pData, is a pointer to the location where the data from the EVF2 file is to be returned. The fourth argument, nSize, is the size in bytes of each element to be read from the file, and the fourth argument, nCount, is the number of elements to read from the file.

The calls to the function pointed to by this function pointer are embedded in the code for EVF2VectorDelivery. The function EVF2VectorDelivery knows the size of the elements and the number of elements to read with each call to this function, so all the user must provide is a function (like FileCallback, previously referenced) which, when called, retrieves nCount elements of size nSize from the location pointed to by pUser1 (or from some constant, hard-coded location) and returns them in the location pointed to by pData. Memory allocation is handled in EVF2VectorDelivery.

The second pointer, pUser2, is not used in EVF2VectorDelivery. Its intended use is as a location to which error messages can be directed, but it can be used in any way the system programmer chooses. It should also be noted that the first pointer, even though its intended use is as a location from which to read the EVF2 data, need not be used this way. For example, if the location of the EVF2 data in memory is always the same, this location can be hard-coded into the file-reading function and the first pointer can be used for anything the system programmer desires, or not used at all. All EVF2VectorDelivery cares about is that the user's function returns the correct amount of data and places it in the passed memory location.

6.1.2.2. Error Handler

The second argument to EVF2VectorDelivery is a pointer to an error handler function. An example of an error handler function used in the SCANSTAEVK demonstration kit software is:

```
int ErrorCallback(const char *pszText,void *pUser1,void *pUser2);
```

The first argument to this function, pszText, is an error message supplied by EVF2VectorDelivery. The other two arguments, pUser1 and pUser2, are the two pointers passed in to EVF2VectorDelivery. As mentioned previously, the first pointer is intended to point to the location of the EVF2 file. In this function, it is used to identify the file that created the error. The second argument is intended to be the location to which the error message is directed. In this function, this is a pointer to the top window of the vector delivery application running on the PC. As a reminder, however, these pointers can be anything the system programmer chooses.

EVF2VectorDelivery calls this function, if it is supplied, with an appropriate text error message when incorrect data is read from the EVF2 file. This function need not be supplied by the user if error messages are not desired. If the function is not supplied, the corresponding argument to EVF2VectorDelivery should be NULL.

6.1.2.3. Fail Handler

The third argument to EVF2VectorDelivery is a pointer to a fail handler function. This function is called by EVF2VectorDelivery when a test failure is detected, i.e., when the test data received from the scan chain does not match the expected test data. An example of a fail handler function used in the SCANSTAEVK demonstration kit software is:

```
int FailureCallback(unsigned long dwSeq,void *pUser1,void
*pUser2);
```

The pointer arguments to this function are the same as those previously discussed and they can be used in the same way. EVF2VectorDelivery supplies the first argument, dwSeq, and it is the sequential number of the test vector being executed when the failure was detected. This may provide valuable information for debugging a board problem.

This function need not be provided if a failure indication is not required. If the function is not provided, then the pointer passed in to EVF2VectorDelivery should be NULL.

6.1.2.4. Register Trace Handler

The fourth argument to EVF2VectorDelivery is a pointer to a register trace handler function. This function need not be provided if register write tracing is not required. If the function is not provided, then the pointer passed in to EVF2VectorDelivery should be NULL.

This function is called by EVF2VectorDelivery whenever the EVF2 file specifies a register write. The prototype of the function used in the SCANSTAEVK demonstration kit is:

```
void TraceCallback(void *pUser1,void *pUser2,unsigned short
wId,unsigned long dwValue,unsigned long dwMasked);
```

The pointer arguments to this function are the same as those used previously. The parameter wId is the index of the SCANSTA101 STA master register to be written. The parameters dwValue and dwMasked are the bits and mask bits to write to the register, respectively. Bits for which the mask is not set to "1" retain their original value after a register write.

It should be noted that this use of a mask is different from the use of a mask in comparing TDO data to an expected value. This mask is used to specify which register bits to write and which to leave at their original values.

6.1.2.5. Debug Handler

In the release version of EVF2VectorDelivery the debug handler function is never called. It is intended to be used to debug the embedded vector delivery process, so it can be defined as the user desires. Since the user has access to the source code, debugging calls can be added as needed and then removed when they are no longer necessary. It is recommended that conditional compilation be used for this purpose.

The parameters of the debug handler function can be set to whatever the user needs in order to accomplish the required debug. In essence, this argument is a placeholder for a generalized debug function pointer to be defined and supplied by the user.

6.1.2.6. Polling Handler

The sixth argument to EVF2VectorDelivery is a pointer to the polling handler function. This function, if it is supplied, is called before every register write during execution of a load-on-the-fly test sequence. This function need not be supplied and, if it is not, the pointer passed to EVF2VectorDelivery should be NULL.

If the function is supplied, its purpose is to permit the system controller to respond to other events in the system during the time when it is supervising a built-in self-test. This applies to a load-on-the-fly test sequence and to a test sequence in which multiple vectors are used with the SCANSTA101 STA master sequencer.

For a load-on-the-fly test sequence, the system controller loads every vector into the SCANSTA101 STA master in real time as opposed to loading all the vectors into the device memory and then initiating the test. This can block execution of other threads in the system controller's program for a significant period of time, since the system controller must continuously update the SCANSTA101 STA master with new vectors. The same thing can happen when the SCANSTA101 STA master sequencer is used. When the polling handler function is called, it can check for events produced by other threads in the system controller program and respond to those events before returning to the self-test thread.

In the SCANSTAEVK demonstration kit used for the case study in this design guide, the polling handler function was written to examine the message queue for messages from all the windows in the vector delivery program and to dispatch these messages. As mentioned, this function is only called during load-on-the-fly vector operations or operations where the SCANSTA101 STA master sequencer is used. The prototype for the function used in the SCANSTAEVK demonstration kit software is:

```
void PollingCallback(void *pUser1,void *pUser2,unsigned long
dwDelay);
```

The first two arguments to this function are the same pointers used in all the other function pointer argument lists. The last argument, dwDelay, is a delay setting in ms which specifies the minimum period the function should wait to detect events from other threads before returning to the load-on-the-fly vector sequence. In the production version of EVF2VectorDelivery, this is hard-coded to zero for each register write in the load-on-the-fly sequence, but it could be edited by the user to be any desired value. For the sequencer operation, the delay time is set by the parameter dwLatencyMs.

6.1.2.7. Timeout Value

When the sequencer is in use, a timeout value can be provided in dwTimeoutMs. This is the maximum amount of time in ms to wait while waiting for the status register in the SCANSTA101 STA master to indicate that the test sequence is complete. A value of zero will produce an infinitely long wait.

This timeout operation can be modified by the user to meet the needs of a particular system. As currently implemented it is a simple timeout, but more complex operation might be required in some systems.

6.1.2.8. User Pointer Arguments

The intended use of the user pointer arguments passed to EVF2VectorDelivery has been described. However, these pointers may be used in any way the system controller programmer desires, so their definitions are flexible.

The system controller programmer should also keep in mind that the source code for EVF2VectorDelivery is provided for inclusion in the system controller program, so it can be changed if desired. For example, the system programmer might choose to pass a third pointer to EVF2VectorDelivery to use in place of (or in addition to) the two pointers in the original argument list. A third pointer might be useful, for example, to pass a memory location for register tracing which might be different from the location for error messages.

The code for EVF2VectorDelivery is designed to provide flexibility for the system programmer. In order to use the function as it is delivered, all that is necessary on the EVF2 interface side is to provide a function that can deliver the data contained in the EVF2 file when the EVF2VectorDelivery function requests it.

6.2. Interface to the SCANSTA101 STA Master Registers

The first task of EVF2VectorDelivery is to read, with the assistance of the functions passed in its argument list, the data from the EVF2 file stored in the system memory. The second function is to perform the indicated reads from and writes to the appropriate SCANSTA101 STA master registers.

The functions required for reading and writing the registers are not optional, of course. The National software assumes that these functions have the prototypes shown in the following listing (which is repeated here for convenient reference) and that the macros shown are directed to those functions. The 16-bit read and write functions are used for register access and the 32-bit versions are used to write to and read from the internal long word memory of the SCANSTA101 STA master.

```
/*
** Prototypes for the specific i/o functions that are currently
**   implemented in another module and resolved during final linkage.
** These could be replaced or removed depending upon the transport
** mechanisim used to access the STA101.
*/
void Write16BitRegister(unsigned short wAddr,unsigned short wData);
void Read16BitRegister(unsigned short wAddr,unsigned short *pwData);
void Write32BitRegister(unsigned short wAddr,unsigned long dwData);
void Read32BitRegister(unsigned short wAddr,unsigned long *pdwData);

/*
** These macros are now defined for the a specific hardware based
** implementation. Simply define these macros to use the appropriate
** i/o functions for the hardware in use.
*/
#define READ16REG(addr,wval)  Read16BitRegister((addr),&(wval))
#define WRITE16REG(addr,wval) Write16BitRegister((addr),(wval))
#define READ32REG(addr,lval)  Read32BitRegister((addr),&(lval))
#define WRITE32REG(addr,lval) Write32BitRegister((addr),(lval))
```

The macro definitions make it easy for the user to change the names of the functions if desired. Only the prototypes and the macro definitions must be changed if this is desired. The changes required are limited to a small section of the code. The user has access to the entire source code, so other more extensive changes could be made if necessary.

The user must supply the functions for reading and writing 16-bit registers and 32-bit memory locations. These functions can do anything else the user needs them to do as long as they eventually read from or write to the appropriate registers in the SCANSTA101 STA master. Address translation, for example, could be handled in these functions.

All of the software supplied by National, including the SVF to EVF2 conversion software and the embedded source code, is designed to make it easy for the user to implement built-in self-test. In many cases National's software can be used as supplied and the user need not be concerned about the details of the built-in self-test. For the purposes of this design guide, however, what the built-in self-test really does at a bit level is illustrated. The next section looks in some detail at how the built-in self-test was implemented.

7. Dissecting the Built-In Self-Test

To really understand how the built-in self-test is implemented, it is useful to examine it in detail on two levels. First, the bit patterns that are to be shifted into the TAP and what they are intended to do are seen in the SVF file. Then, by decompilation and examination of the EVF2 file, the details of the mapping from SCANSTA101 STA master register reads and writes onto the bit patterns from the SVF file are shown.

7.1. The SVF File

The SVF files produced by each tool were somewhat different, although they all achieved the same objective. The beginning sections of the SVF file produced by the Corelis ScanExpressTPG tool are shown below.

```
!-------------------------------------------------------------------
----------
!
! Generated by Corelis CVFtoSVF Converter Version 1.07
!
! Generated from C:\Documents and Settings\CJRJSC\My Documents\
Scan\ScanExpress_Projects\SCANSTAEVK_Demo\JTAG_EVK_Demo_
interconnect_ic.cvf
! CVF File Version: 1.01
! CVF Test Name    : JTAG_EVK
! CVF Test Type    : INTERCON
! CVF Revision     : 1.01
! CVF File Date    : 091009
!
!-------------------------------------------------------------------
----------
!
!            SVF FILE STATEMENTS OPCODE SYNTAX SUMMARY
!            --- ---- ---------- ------ ------ -------
!
! ENDDR   - Specify the end state for any data register (SDR) scan
operation
!
! ENDIR   - Specify the end state for any instruction register
(SIR) scan
!           operation
!
! RUNTEST - Forces the IEEE 1149.1 bus to the specified run state
for
!           a specified number of clocks. This can be used to
control RUNBIST
!           operation in the target
!
! SDR     - Performs an IEEE 1149.1 Data Register scan
!           Shift data opcode, followed by number-of-bits
(decimal),
!           serial-data-out (hex), expected-serial-data-in (hex),
mask-of-
!           serial-data-in (hex)
!
! SIR     - Performs an IEEE 1149.1 Instruction Register scan
!           Shift instruction opcode, followed by number-of-bits
(decimal),
!           serial-data-out (hex), expected-serial-data-in (hex),
mask-of-
!           serial-data-in (hex)
!
! STATE   - Move the boundary scan controller state-machine to this
stable
!           state
!
! TRST    - Controls the optional Test Reset line
!
!-------------------------------------------------------------------
----------
!
!           NOTE: Text comments are marked with a preceding
'!' character
!           NOTE: MASK bit value of '0' masks out the
relevant TDO bit
!           NOTE: Serial-data-out (TDI) is shifted out with
LSB first
!
!-------------------------------------------------------------------
----------

TRST OFF;

STATE IDLE;

!-------------------------------------------------------------------
----------
!
! Source CVF file generated by ScanPlus TPG Version 2.03
!
!-------------------------------------------------------------------
----------

SIR         8  TDI (09)
               TDO (00)
              MASK (00);

SIR         8  TDI (8E)
               TDO (25)
              MASK (FF);

SDR         8  TDI (01)
               TDO (00)
              MASK (00);

SIR         8  TDI (E7)
               TDO (25)
              MASK (FF);

!-------------------------------------------------------------------
----------
!
! TOPOLOGY INFORMATION
! The devices are listed in the order from TDI to TDO of the
board
!
!   STA_U1       (SCANSTA111 Enhanced Scan Bridge at address
0x00000009 on TAP0)
!                              - Instruction Length ( 8)
Boundary Length (  1)
!   U1           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 19)
!   U2           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 18)
!   PAD          (on TAP0) - Instruction Length ( 1) Boundary
Length (  1)
!
!-------------------------------------------------------------------
----------

SIR        25  TDI (1FF0505)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (6DFAAFFFFF)
               TDO (0000000000)
              MASK (0000000000);

SIR        25  TDI (1FE0001)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (4EFB3FFFFF)
```

```
                TDO  (00E807FC96)
               MASK  (00E807FF96);

SDR         39  TDI  (77FC3FFFFF)
                TDO  (00E807FE96)
               MASK  (00E807FF96);

SDR         39  TDI  (47FFCFFFFF)
                TDO  (00E807FE96)
               MASK  (00E807FF96);

.
.
.

! Total number of vectors : 21
```

Comments in this file are preceded by an exclamation point.

The TRST OFF and STATE IDLE instructions in the file already have been discussed. Next, the first Scan Instruction Register (SIR) instruction is examined and is shown as:

```
SIR          8  TDI  (09)
                TDO  (00)
               MASK  (00);
```

This is a SIR instruction. That means that when the system executes it, the TAP will be sent to the Shift-IR state prior to scanning in the data on the TDI input. The data will go into the instruction register. At the beginning of the test sequence, since there is a SCANSTA111 multiplexer in the system and since it is in the Wait-for-Address state, the scan chain consists only of this device; so the instruction register chain is the instruction register in the SCANSTA111 multiplexer, which is 8 bits long.

The eight bits scanned into the TDI input of the SCANSTA111 multiplexer, which subsequently pass into its instruction register, are 0x09 or b00001001. This is the address of the SCANSTA111 multiplexer which is set by the static bit values on its address inputs, S6:S0. The SCANSTA111 multiplexer has a seven-bit address space. The ATPG software was told that the address of the SCANSTA111 multiplexer was set to 0x09 prior to generating the test vectors. Accordingly, the first instruction in the SVF file selects the SCANSTA111 multiplexer by scanning its address into its instruction register. When the SCANSTA111 multiplexer detects that it has been addressed, it enters the Run-Test-Idle state and waits for the system to configure it.

The next SIR instruction looks like this:

```
SIR          8  TDI  (8E)
                TDO  (25)
               MASK  (FF);
```

Since the SCANSTA111 multiplexer has not been configured yet, the instruction register chain is still just the 8-bit instruction register of the SCANSTA111 multiplexer. The instruction scanned in at this step is 0x8E or b10001110. The SCANSTA111 multiplexer datasheet illustrates that this instruction is MODESEL, which puts the Mode Register 0 in the data path.

The TDO value to be compared is 0x25 or b00100101. Referring to the SCANSTA111 multiplexer datasheet, it is evident that upon exiting the Capture-IR state, the value bXXXXXX01 is captured into the instruction register, where the six Most-Significant Bits (MSB) are the values set on the address lines S5:S0. Since the address lines are set to 0x09, the instruction register should capture b00100101 and this should be the value shifted out of the TDO output. This corresponds to the expected TDO value of 0x25.

The mask is set to 0xFF or b11111111. This indicates that all the bits received on the TDO output of the SCANSTA111 multiplexer should be compared to the expected value 0x25. In this case, that is the appropriate behavior.

Capturing and testing the TDO output after this SIR instruction provides a valuable confirmation that the JTAG TAP is configured and operational. Obviously, if the JTAG TAP is not working, nothing else can be tested in the system. The SVF files produced by the various ATPG tools specify testing the operation of the JTAG TAP in different ways. All of the JTAG tools produce SVF files which include provisions for verifying the operation of the JTAG TAP.

The next instruction is the SVF file sets up the SCANSTA111 multiplexer mode register:

```
SDR          8  TDI  (01)
                TDO  (00)
               MASK  (00);
```

This is a SDR instruction, so the SCANSTA111 multiplexer TAP, which is still the only thing in the scan chain, will be put into the Shift-DR mode before the data is shifted into the TDI input. Since the previous instruction issued was the MODESEL command, the data shifted into the TDI input goes into the Mode Register 0. The SCANSTA111 multiplexer datasheet shows that setting the Mode Register 0 value to 0x01 sets the scan chain configuration as follows:

$TDI_B \rightarrow$ Register $\rightarrow LSP_0 \rightarrow$ PAD $\rightarrow TDO_B$

In other words, the data shifted into the TDI_B input will go first to whatever register is inserted in the scan chain in the SCANSTA111 multiplexer, then to local scan port 0 and

whatever is on the scan chain attached to local scan port 0, then to a pad bit register in the SCANSTA111 multiplexer which re-synchronizes the local scan port, then finally to the TDO_B output. The ATPG software was told that the JTAG devices were all on a chain attached to LSP_0 when the SVF file was generated. This scan chain configuration will be active after the UNPARK instruction is issued to the SCANSTA111 multiplexer.

The current mode register contents are shifted out on the TDO output when the new mode register contents are shifted in, but these are not interesting, so the mask is set to 0x00. This means that no comparison will be performed on the TDO output data at this step.

The next instruction in the file looks like this:

```
SIR         8  TDI (E7)
               TDO (25)
              MASK (FF);
```

Again, this is a Scan-Instruction-Register instruction, so the TAP for the scan chain (which still consists, at this point, only of the SCANSTA111 multiplexer) will be put into the Shift-IR state before the data is shifted out to the TDI input. Instruction 0xE7, or b11100111, is the op-code for UNPARK, which puts the desired local scan port into the scan chain. As before, the instruction register captures 0x25 and the value scanned out of the TDO output is compared to the expected value with a mask of 0xFF.

Following this UNPARK instruction, the scan chain now includes everything on local scan port 0 in addition to the SCANSTA111 multiplexer itself. The next instruction in the file looks like this:

```
SIR        25  TDI (1FF0505)
               TDO (0000202)
              MASK (0000606);
```

This is another SIR instruction, but now the data pattern is 25 bits long instead of just eight. That is because the 8-bit instruction registers of the SCAN921224 deserializer and the SCAN921023 serializer are now in the scan chain in addition to the 8-bit instruction register of the SCANSTA111 multiplexer, and there is a pad bit at the end of the scan chain. What this instruction does is discussed next.

The bit pattern and the way it divides among the various registers is shown in ***Figure 7-1***. What this instruction in the SVF file does is (1) sets the SCANSTA111 multiplexer to BYPASS mode, inserting its one-bit bypass register in the scan chain; and (2) issues the SAMPLE instruction to the SCAN921224 deserializer, which puts it in a state to preload the next data shifted into the boundary register; and (3) issues the SAMPLE instruction to the SCAN921023 serializer. For the SCAN921224 deserializer and the SCAN921023 serializer, the SAMPLE and PRELOAD instructions are the same.

Once this instruction is executed, which occurs when the TAP state machine enters the Update-IR state, the scan chain will consist of the following: the SCANSTA111 multiplexer's bypass register (one bit), the SCAN921224 deserializer's boundary register (19 bits), the SCAN921023 serializer's boundary register (18 bits), and the pad bit inserted by the SCANSTA111 multiplexer, which is the same pad bit that was inserted in the scan chain when the Scan-Instruction-Register instruction was executed. The scan chain at this point will look like a 39-bit shift register. The next data scanned into the TDI port by a Scan-Data-Register instruction will go into this shift register.

The expected TDO output is a 25-bit pattern consisting of 0x00000202, and the mask is a 25-bit pattern consisting of 0x00000606. This means that only bits 1, 2, 9, and 10 of the output from the TDO will be compared, and the expected bit pattern will contain ones at bits 1 and 9 and zeros at bits 2 and 10. As seen in ***Figure 7-1***, the comparison is meant to look for the required 01 pattern in the last two bits of the instruction register for the SCAN921224 deserializer and the SCAN921023 serializer, and nothing else.

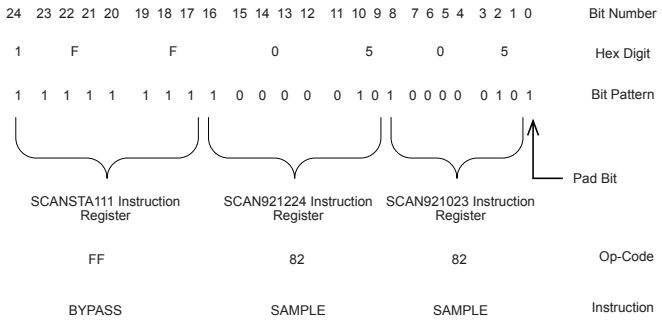

Figure 7-1. Instruction Bit Pattern and Decomposition

The next instruction in the file loads the initial data into the boundary registers of the SCAN921224 deserializer and SCAN921023 serializer as follows:

```
SDR        39  TDI (6DFAAFFFFF)
               TDO (0000000000)
              MASK (0000000000);
```

The bit pattern is 39 bits long; that is, 1 bit for the SCANSTA111 multiplexer bypass register plus 19 bits for the SCAN921224

deserializer boundary register plus 18 bits for the SCAN921023 serializer boundary register plus 1 pad bit. This bit pattern can be examined according to the scheme shown in ***Figure 7-1***. Bit 0, which is the pad bit, can be ignored. As shown, the contents of the boundary register for the SCAN921023 serializer are b111111111111111111 – all ones. If the boundary register for the SCAN921023 serializer is examined, it is evident that it only has one output cell in its boundary register, bit 5, the differential output. This boundary cell is controlled by bit 4, its enable bit. So this SAMPLE/PRELOAD instruction enables the single output boundary cell on the device and sets its value to logic 1. This value will be presented at the output pin of the SCAN921023 serializer when an EXTEST instruction is executed. When the EXTEST instruction is executed, the values on the input boundary cells will be replaced by the values sampled at the input pins of the SCAN921023 serializer.

Also shown is that the contents of the boundary register for the SCAN921224 deserializer are b1011011111101010101. The SCAN921224 deserializer has more output boundary cells than the SCAN921023 serializer. For the SCAN921224 deserializer, bits 1, 2, 3, 4, 5, 6, 7, 13, 14, 16, 17, and 18 in the boundary register are all output boundary cells. All of these cells except bit 14 are controlled by bit 0 in the boundary register. This bit is set to logic 1 in the current bit pattern, meaning that all these cells are enabled.

The output cell on bit 14, which is the LOCK* output, is controlled by bit 15 in the boundary register, which is also set to logic 1 in this bit pattern. So output cell 14 is also enabled. In this bit pattern, bit 14 is logic 0, so the output of the boundary cell on bit 14 will be logic 0 when the EXTEST instruction is executed.

Bits 16, 17, 18, 1, 2, 3, 4, 5, 6, and 7, in that order, are the receiver output bits 9:0. From the bit pattern, it is evident that these outputs are driven by a bit pattern 1010101010 from Most-Significant Bit (MSB) to Least-Significant Bit (LSB). The only other output bit in the boundary register is bit 13, the RCLK output bit. This bit is driven to logic 1 in the pattern shown previously.

The MSB of the entire bit pattern is a logic 1. This goes into the bypass register of the SCANSTA111 multiplexer, but it will not be visible to the rest of the circuit.

At this point, the boundary registers of the SCAN921224 deserializer and SCAN921023 serializer are preloaded with the first data pattern to be presented at their output pins. This data pattern is presented at the output pins and the input pins are sampled when the EXTEST instruction is executed.

The next line in the SVF file is:

```
SIR        25  TDI (1FE0001)
                TDO (0000202)
               MASK (0000606);
```

The expected TDO output and the mask are the same for this Scan-Instruction-Register instruction as for the previous one. ***Figure 7-2*** shows the instruction. As the figure indicates, the SCANSTA111 multiplexer is left in BYPASS mode and the EXTEST instruction is issued to both the SCAN921224 deserializer and the SCAN921023 serializer. This will cause each device to assert the preloaded bit pattern on its output cells and to sample the incoming bit pattern on its input cells. This will be shifted out of the TDO output when new data is loaded into the TDI input at the next step.

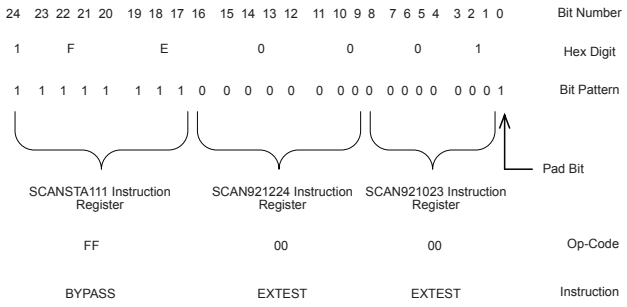

Figure 7-2. Instruction Bit Pattern and Decomposition

The next step in the SVF file is:

```
SDR        39  TDI (4EFB3FFFFF)
                TDO (00E807FC96)
               MASK (00E807FF96);
```

Upon examination of the mask and expected data for this instruction, the bits 0 and 38 of the mask are both 0, meaning that neither the pad bit from the TDO output (bit 0) or the bit placed in the bypass register of the SCANSTA111 multiplexer (bit 38) will be considered in comparing the received output with the expected output.

For the SCAN921224 deserializer, the mask is given by bits 19-37 of the bit pattern above. This bit pattern is b0000001110100000000. This means that only bits 8, 10, 11, and 12 of the boundary register will be used in the comparison. These bits are the same in the expected TDO pattern, so all of them should be 1. From the boundary scan register description for the SCAN921224 deserializer, only bits 8, 9, 10, 11, and 12 are input bits. In the SCANSTAEVK SerDes board schematic, the boundary cell on bit 9, which is the REFCLK input, is not connected to anything that can drive a logic level on it (the TCLK

pin it is connected to on the SCAN921023 serializer is also an input). So this bit is masked out in the comparison since it is not known what logic level to expect at this input.

In the SCAN921224 deserializer, boundary cell bits 8, 11, and 12 correspond to the RCLKR/F, PWRDN, and REN inputs respectively, and all these are tied high through S3. The ATPG tool was told that all the switches in DIP switch S3 were closed, so it was able to correctly determine that there should be static high levels on these pins.

Bit 10 of the boundary cell is the differential input, which is connected to the differential output of the SCAN921023 serializer. The ATPG tool was told that these two pins are connected since this is not indicated on the netlist. Accordingly, it was able to determine that the logic 1 driven on the differential output of the SCAN921023 serializer should be detected at the differential input of the SCAN921224 deserializer.

For the SCAN921023 serializer, the mask is given by bits 1-18 of the previous bit pattern. This bit pattern is b111111111111001011. For the SCAN921023 serializer, all the bits in the boundary register are input bits except bits 4 and 5, which are masked off in the mask shown previously. Bit 2 is also masked off because this is the TCLK input line, which, as noted, does not have an accessible driver. Therefore, this input is not considered in the comparison.

For the SCAN921224 deserializer, the mask and the expected data were exactly the same, meaning that all the bits to be considered in the comparison were expected to be at logic 1. For the SCAN921023 serializer, this is almost true. The only bits that are different in the mask and the expected data are bits 7 and 8. Bit 7 is the SYNC1 input bit. Referring to ***Figure 3-7***, it is clear that this bit is tied low through R1. The ATPG tool was instructed to sense a static low on this input, which explains the resulting pattern.

Bit 8 is the SYNC2 input bit which is connected to the LOCK* output bit of the SCAN921224 deserializer. In the ***Figure 3-7*** schematic, the connection is made by a jumper between pins 1 and 2 of header J17. It is important to remember that this bit was driven low by the preloaded pattern, bit 14 of the boundary register of the SCAN921224 deserializer, so a low should be sensed when the EXTEST is performed. This is what is shown in the comparison pattern.

For the SCAN921023 serializer, bits 0, 17, 16, 15, 14, 13, 12, 11, 10, and 9, in that order, are digital input bits DIN9:0. In ***Figure 3-7***, these are all tied high through resistor networks RN1, RN2, and RN3. The ATPG tool was instructed to sense static high levels at all these inputs. This is what is seen in the previous bit pattern.

Bits 1, 3, and 6 of the SCAN921023 serializer boundary register are the TCLKR/F, DEN, and PWRDN bits respectively, and all these are tied high through switch S2. The bit pattern specifies sensing static high levels on these bits.

So this mask and expected bit pattern tests the static input levels and partially tests the connections between the SCAN921224 deserializer and the SCAN921023 serializer. This is exactly what was desired from the built-in self-test.

As the data from the previous test is read, the data is shifting in for the next test. The pattern to be shifted into the TDI input is 0x4EFB3FFFFF, and the pattern is 39 bits long. Again, bit 0 is the pad bit and bit 38 is the bit for the bypass register of the SCANSTA111 multiplexer, so neither of these bits will be involved in the test.

The bit pattern shifted into the boundary register of the SCAN921023 serializer is the same as that from the previous test, b111111111111111111. Again, the only output boundary cell in the SCAN921023 serializer is bit 5 of the boundary register, which is the differential output. This output cell is driven to logic 1 for this test, the same as it was for the previous test.

The bit pattern shifted into the boundary register of the SCAN921224 deserializer is different from that used in the previous test. This bit pattern is b0011101111101100111. Bit 15 enables bit 14. In this test, bit 14, which is the LOCK output, is enabled and is set to 1, where it was set to 0 in the previous test.

Bit 0, also set to 1, enables all the other output bits. There is no way for the boundary scan built-in self-test to directly sense the values on ROUT9:0, but the test sets them anyway. They are set to a bit pattern of 1001100110, a different pattern from that used in the previous test. These output bits, even though they cannot be directly sensed, are useful for detecting shorts between the traces on the board.

Bit 13, the RCLK output, is set to 0. It was set to 1 in the previous test.

Review of one more instruction in the SVF file in detail is useful. The next line of the file is:

```
SDR        39   TDI  (77FC3FFFFF)
                TDO  (00E807FE96)
               MASK  (00E807FF96);
```

The mask is the same for this instruction as for the previous one. In fact for the remainder of the SVF file, the mask is the same. This means that comparison will be made between the same bits in the received data pattern, the bits for which the expected value is known.

Also, the expected bit pattern for this instruction is almost the same as the previous one. In fact, for the SCAN921224 deserializer, the expected bit pattern is exactly the same. That is because all the input cells in the boundary register are connected to static logic levels that will not change during the test except for bit 10, the differential input, which is connected to the differential output of the SCAN921023 serializer. Since this differential output is still driving a 1, just like it was in the previous test, exactly the same data pattern on the SCAN921224 deserializer's input cells is expected.

For the SCAN921023 serializer, the only difference in the expected bit pattern is that bit 8, which was previously a 0, is now a 1. This bit is the SYNC2 input, which is connected to the LOCK* output of the SCAN921224 deserializer. In this test, the LOCK* output of the SCAN921224 deserializer is driven to a logic 1, so this is the bit that is expected at the input of the SCAN921023 serializer. All the other input bits are driven to static logic levels, so the expected bit pattern on those bits does not change.

The balance of the SVF file, which is included in its entirety in Chapter 9, can be examined in the same way, and where the expected bit pattern changes will be evident. The mask is the same for all the tests in the file. The expected bit pattern only changes in two bits. The differential input of the SCAN921224 deserializer, bit 10 in the boundary register for this device or bit 29 in the TDO bit sequence, changes when different values are driven on bit 5 of the boundary register for the SCAN921023 serializer. The SYNC2 input of the SCAN921023 serializer, bit 8 of the boundary register for the SCAN921023 serializer or bit 9 in the TDO bit sequence, changes when the LOCK output of the SCAN921224 deserializer changes. This is bit 14 of the SCAN921224 deserializer boundary register.

Nothing else in the expected bit pattern changes. The output bits that cannot be sensed are driven to different values, but this does not change the expected bit pattern. Only the output bits that can be sensed on an input bit change in the expected bit pattern.

This SVF file is for built-in self-test of a very simple scan chain with only three devices. For an operational board, the scan chain might be much longer and the SVF file considerably more complex. Crafting a test sequence manually, without the aid of an ATPG tool, to test a more complicated board would be time-consuming and prone to errors. This is the value of the ATPG tools. They are capable of generating exhaustive boundary scan test sequences from relatively simple inputs.

Normally it is neither necessary nor desirable (and maybe not even possible) to dissect a SVF file as has been done here. It is instructive, however, to perform this exercise at least once in order to gain an appreciation for what the test sequence is really accomplishing.

One interesting point about this test sequence is that many of the test steps don't appear to do anything because they involve manipulation of outputs that cannot be sensed. These test steps are useful, however, because they can detect the presence of solder bridges on the board. If an output change which is not supposed to change the expected bit pattern does change it, then something is wrong with the board. The tests where the expected bit pattern does not change are designed to detect these problems.

It should be noted that this is not the only test sequence that will work for this board. As an example, the following fragment of the SVF file created by the JTAG Technologies ProVision tool is considered.

```
! SVF File: "c:\Bst32\ProVision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.svf"
! Input GEN File: "c:\bst32\provision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.gen"
! Input APL File: "c:\Bst32\ProVision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.apl"
! Input CON File: "c:\bst32\provision\projects\JTAG_EVK_Demo\
interconnect_JTAG_Tech\interconnect_JTAG_Tech\interconnect_
JTAG_Tech.con"

FREQUENCY 25000000 Hz;

STATE RESET;
SIR 8
 TDI  (09);
SIR 8
 TDI  (8e);
SDR 8
 TDI  (01);
SIR 8
 TDI  (e7);
!
! Load Sample/Preload Instruction
!
SIR 25
 TDI  (1550504);
SDR 70
 TDI  (000000003f07f80060);
!
! Load Test Instruction
!
SIR 25
```

```
 TDI   (1540000);
SDR 70
 TDI   (000000000400080020)
 TDO   (03f03c07c0e8000694)
 MASK (03ffffffc0e8004694);
SDR 70
 TDI   (000000001c00080020)
 TDO   (03f03c07c0c8000494)
 MASK (03ffffffc0e8004694);
```

The first few instructions, the ones that select the SCANSTA111 multiplexer, select its LSP, and unpark the LSP, are essentially the same in this file as in the previous one. This SVF file does not specify a comparison pattern for the TDO data, so no mask is necessary.

Next, the Sample/Preload instruction called out in this SVF file is considered. In this SVF file, this instruction is:

```
SIR 25
 TDI   (1550504);
SDR 70
 TDI   (000000003f07f80060);
```

This instruction is different from that shown in the previous SVF file. The instruction details are shown in ***Figure 7-3***. As the figure indicates, this SVF file sends the SCANSTA111 multiplexer the IDCODE instruction instead of the BYPASS instruction. This means that when data is scanned in (the SDR instruction that occurs next in the file, for example), the SCANSTA111 multiplexer's ID register will be inserted in the scan chain and the ID code for the SCANSTA111 multiplexer will be scanned out as the test data is scanned in.

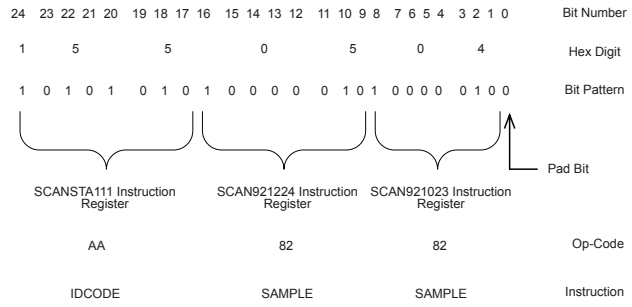

Figure 7-3. Instruction Bit Pattern and Decomposition

The next instruction in the file, which scans the IDCODE instruction into the SCANSTA111 multiplexer and the EXTEST instruction into both the SCAN921023 serializer and the SCAN921224 deserializer is:

```
SIR 25
 TDI   (1540000);
SDR 70
 TDI   (000000000400080020)
 TDO   (03f03c07c0e8000694)
 MASK (03ffffffc0e8004694);
```

On this data register scan and on each subsequent data register scan, the 32 most-significant bits of the TDI bit pattern are all zeros. This bit pattern is scanned into the SCANSTA111 multiplexer's data register. Since the SCANSTA111 multiplexer has been sent the IDCODE instruction, the data register is the ID code register. The data that comes out of the SCANSTA111 multiplexer is the 32-bit ID code, which is 0x1FC0F01F (this comes from the SCANSTA111 BSDL file).

This is the reason that the bit pattern in this file is 70-bits long as opposed to the 39-bit long patterns shown in the previous SVF file. The TDI and TDO bit sequences in this case are targeted at a shift register consisting of the 32-bit ID register in the SCANSTA111 mulitplexer plus 19 bits for the SCAN921224 deserializer boundary register plus 18 bits for the SCAN921023 serializer boundary register plus 1 pad bit.

Next, the 32 most-significant bits of the TDO bit pattern and the mask is considered.The 32 most-significant bits of data scanned out from the scan chain should contain the SCANSTA111 multiplexer's ID code. The least-significant 28 bits of the 32-bit ID code are compared to the known value obtained from the BSDL file. The four most-significant bits are the revision code, which is not compared.

For the TDO comparison value, the most-significant 32 bits of the 70-bit value are 0x0FC0F01F. The most-significant 32 bits of the mask are 0x0FFFFFFF. From this, it is evident that the comparison is performed only on the least-significant 28 bits of the ID code for each test vector.

Using the IDCODE instruction to command the SCANSTA112 multiplexer to scan out its ID code on each data vector verifies the operation of the JTAG TAP. In this SVF file, every shift operation incorporates a test of the operation of the SCANSTA112 multiplexer. As mentioned, each of the ATPG tools produces a SVF file that includes verification of the operation of the JTAG TAP. The IDCODE instruction is the method used in this SVF file to verify the operation of the JTAG TAP.

The remainder of the SVF file consists of test data scanned into the scan chain and comparison values and masks to check the data scanned out of the scan chain. The bit sequences in this SVF file check the same interconnect errors as those in the previous file. When this file was converted into an EVF2 file and deployed to the SCANSTAEVK demonstration system, it was able to detect all the induced board faults just as the previous SVF file was.

7.2. The EVF2 File

The SVF file describes the boundary scan tests in a human-

readable format. In order to implement the required boundary scan test with the SCANSTA101 STA master, the SVF file test sequence must be converted into a series of register writes and reads to the SCANSTA101 STA master that will result in the performance of the desired tests.

National's EVF Workbench graphical conversion program is used to convert the SVF file into an EVF2 file which can be delivered by the embedded software to the SCANSTA101 STA master. The EVF2 file is in a binary format and is not readable by humans. National provides a utility, Evf2Dump, to convert the EVF2 file into a human-readable format which is useful for examining the mechanics of the test sequence.

The converted EVF2 file corresponding to the first SVF file presented earlier is shown below. Note that converting the EVF2 file into a human-readable format normally is not necessary. This converted file is examined, in this instance, only to describe what the SCANSTA101 STA master is really doing to perform the built-in self-test. Every record in this file will not be examined, only a few records to understand further the implementation of the test sequence.

```
Verbose mode
=============== HEADER ====================
Largest vector bit length: 39 bits
Largest buffer bit length: 0 bits
Macro record count: 4
Vector record count: 23
Buffer record count: 0
Register record count: 28
Assumed initial state: Undefined
Defined final state: Undefined
==========================================
===== REGISTER Sequence # 0 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000004
          Data           = 00000004
          Register bits = .............................1..

===== REGISTER Sequence # 1 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 5 (CLKDIV)
          Mask           = 000000FE
          Data           = 00000020
          Register bits = .......................0010000.

===== REGISTER Sequence # 2 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000003
          Data           = 00000040
          Register bits = .............................00

===== REGISTER Sequence # 3 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 2 (INTCTRL)
          Mask           = 00001E00
          Data           = 00000000
          Register bits = ...................0000.........

===== MACRO Sequence # 0 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 4
          Macro          = 0503007C

===== VECTOR Sequence # 0 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number   = 0
          Macro number  = 4
          Load on fly   = NO
          Compare data  = NO
          Use mask       = NO
          Array Length  = 0 longwords
          Data length   = 0 bits
          Data length   = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 4 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = .............................001

===== REGISTER Sequence # 5 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000008
          Data           = 00000043
          Register bits = .............................0...

===== MACRO Sequence # 1 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 3
          Macro          = 01030000

===== VECTOR Sequence # 1 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number   = 0
          Macro number  = 3
          Load on fly   = NO
          Compare data  = NO
          Use mask       = NO
          Array Length  = 0 longwords
          Data length   = 0 bits
          Data length   = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 6 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = .............................001

===== MACRO Sequence # 2 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
```

```
          Macro number  = 2
          Macro          = D3020330

===== VECTOR Sequence # 2 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number   = 3
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                  DATA[00000000]: 00000009
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000000
          Mask length   = 8 bits
          Mask length   = 1 longwords
                  MASK[00000000]: FFFFFFFF

===== REGISTER Sequence # 7 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 3 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number   = 2
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                  DATA[00000000]: 0000008E
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000025
          Mask length   = 8 bits
          Mask length   = 1 longwords
                  MASK[00000000]: FFFFFF00

===== REGISTER Sequence # 8 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== MACRO Sequence # 3 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 1
          Macro          = D2020320

===== VECTOR Sequence # 4 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 8 ticks
          Slot number   = 3
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                  DATA[00000000]: 00000001
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 9 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 5 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 8 ticks
          Slot number   = 2
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                  DATA[00000000]: 000000E7
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 10 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 6 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 25 ticks
          Slot number   = 1
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 25 bits
          Data length   = 1 longwords
                  DATA[00000000]: 01FF0505
          Expect length = 25 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000202
          Mask length   = 25 bits
          Mask length   = 1 longwords
                  MASK[00000000]: FFFFF9F9

===== REGISTER Sequence # 11 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000002
          Register bits = ............................010

===== VECTOR Sequence # 7 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number   = 3
```

```
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 6 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FAAFFFFF
                   DATA[00000001]: 0000006D
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: 00000000
                 EXPECT[00000001]: 00000000
          Mask length   = 39 bits
          Mask length   = 2 longwords
                   MASK[00000000]: FFFFFFFF
                   MASK[00000001]: FFFFFFFF

===== REGISTER Sequence # 12 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 8 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 25 ticks
          Slot number   = 1
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 1 longwords
          Data length   = 25 bits
          Data length   = 1 longwords
                   DATA[00000000]: 01FE0001
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 13 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000002
          Register bits = ............................010

===== VECTOR Sequence # 9 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 6 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FB3FFFFF
                   DATA[00000001]: 0000004E
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: E807FC96
                 EXPECT[00000001]: 00000000
          Mask length   = 39 bits
          Mask length   = 2 longwords
                   MASK[00000000]: 17F80069
                   MASK[00000001]: FFFFFFFF

===== REGISTER Sequence # 14 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 10 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 4 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FC3FFFFF
                   DATA[00000001]: 00000077
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: E807FE96
                 EXPECT[00000001]: 00000000
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 15 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

.
.
.

DUMP COMPLETE
```

Comparing this file to the EVF2 record specification, it is easy to see that there is a one-to-one correspondence between the two. The header record is self-explanatory. And in the header record there are counts for each of the other record types in the file.

The first few records in the file are register records. These specify data to be written to the control registers of the SCANSTA101 STA master. For example, the first register record is:

```
===== REGISTER Sequence # 0 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000004
          Data           = 00000004
          Register bits = ............................1..
```

This record specifies that bit 2 of the setup register of the SCANSTA101 STA master is to be set while all the other bits in the setup register retain their previous values. Referring to the SCANSTA101 datasheet, setting bit 2 of the setup register resets the SCANSTA101 STA master. This action is automatically inserted by the svf2evf converter before any other register accesses to the SCANSTA101 STA master are performed.

The second register record sets the clock divider register in the SCANSTA101 STA master. The value written to the clock divider register, b0010000, implies that the system clock is to be divided by 32 to produce the test clock.

The third register record clears bits 0 and 1 in the setup register. This sends the SCANSTA101 STA master into normal operation mode.

The remainder of the register records can be interpreted in the same way. Moving on to the first macro record, it is:

```
===== MACRO Sequence # 0 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 4
          Macro          = 0503007C
```

A macro record sets the data for one of the macros available in the SCANSTA101 STA master. In this case, this macro record sets up macro number 4. The macro is 32-bits long. Next, the bit structure of this macro is reviewed.

In ***Figure 7-4***, bits 16 and 17 are considered first. These bits are 11, indicating that this is a STATE macro. In other words, this macro is used to send the SCANSTA101 STA master output TAP controller to a given state.

Bits 24-26 contain the pre-shift TCK_SM count, which is a little misleading since this isn't a shift macro. The value contained there is 5, which really means that when the vector corresponding to this macro is started by writing its vector number into the START register, the SCANSTA101 STA master will send 5 TCK_SM pulses while driving the TMS_SM (Test Mode Select – Scan Master) line with bits 2-6 of the macro. This behavior is described in the SCANSTA101 STA master datasheet.

Bits 2-6 of the macro are all ones, so the effect of this macro will be to output 5 TCK_SM pulses with the TMS_SM line held high. This is a "five-high TMS reset". When this macro is executed, it will send the TAP controllers of all the devices in the scan chain into the Test-Logic-Reset state. That is what this macro is designed to do.

The next record is a vector record which references this macro, macro number 4. The record is:

```
===== VECTOR Sequence # 0 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number   = 0
          Macro number  = 4
          Load on fly   = NO
          Compare data  = NO
          Use mask       = NO
          Array Length  = 0 longwords
          Data length   = 0 bits
          Data length   = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords
```

This is, in some sense, a dummy vector because it does not contain any data and it does not consume any clocks. All it

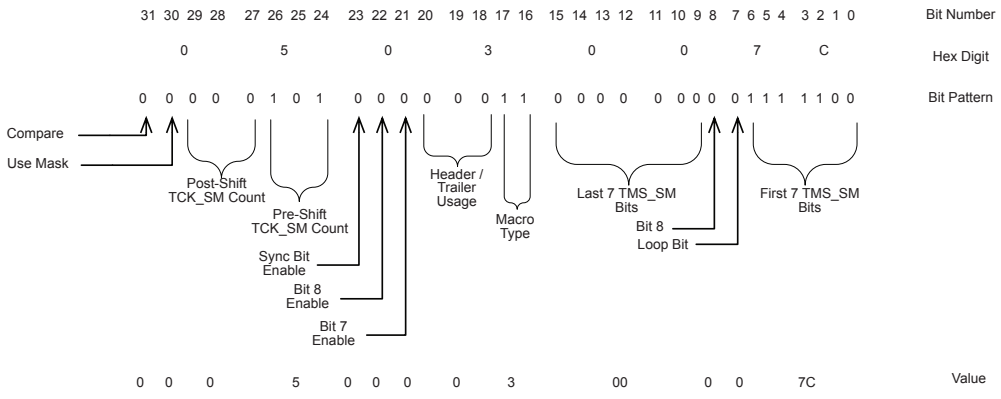

Figure 7-4. Bit Structure of the First Macro Record

really does is reference macro number 4, the STATE macro previously set up. When this vector is started, macro number 4 runs and performs a “five-high TMS reset”.

So the next record is a register record which runs this vector, vector number 1 (vector slot number 0). This record is:

```
===== REGISTER Sequence # 4 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = ............................001
```

When this record executes, it writes a value of 001 to the bottom three bits of the START register. This activates vector number 1 (vector slot number 0), which, in turn, activates macro number 4. The effect of this is to reset the scan chain TAP controllers.

The next record is another register record, shown below.

```
===== REGISTER Sequence # 5 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000008
          Data           = 00000043
          Register bits = ............................0...
```

This record clears bit 3 of the SETUP register, which sets the asynchronous TRST* outputs high, their inactive state. It is important to remember that the first instruction in the SVF file was TRST OFF. This register record implements that SVF command by setting the TRST* line to its inactive state.

The next three records set up and run another macro with a dummy vector. These records are:

```
===== MACRO Sequence # 1 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 3
          Macro          = 01030000

===== VECTOR Sequence # 1 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number   = 0
          Macro number  = 3
          Load on fly   = NO
          Compare data  = NO
          Use mask       = NO
          Array Length  = 0 longwords
          Data length   = 0 bits
          Data length   = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 6 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = ............................001
```

If the macro is examined, it becomes clear that it is also a STATE macro and that it calls for a pre-shift TCK_SM count of 1. The TMS_SM line will be driven to zero during this single clock count. When this macro runs, the TAP controller is in the Test-Logic-Reset state because of the previous macro operation. This macro operation sends it to the Run-Test-Idle state.

The next record is another macro record. This record is:

```
===== MACRO Sequence # 2 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 2
          Macro          = D3020330
```

Figure 7-5 shows the bit structure for this macro record.

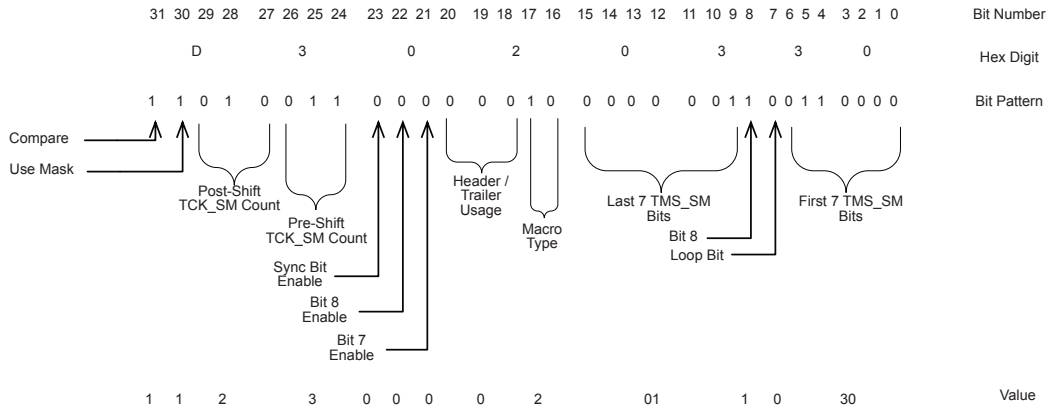

Figure 7-5. Bit Structure of the Second Macro Record

The type of this macro is 10. This is a shift macro with capture. The pre-shift TCK_SM count for this macro is 3, which means that it will output bits 4-6 on the TMS_SM line when it is activated. These bits are 1-1-0.

The TAP controller is in the Run-Test-Idle state when this macro is initiated, so this sequence of TMS_SM bits sends it to the Capture-IR state, ready for instructions to be shifted into the instruction register. These instructions are contained in the following vector record.

```
===== VECTOR Sequence # 2 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number    = 3
          Macro number   = 2
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 3 longwords
          Data length    = 8 bits
          Data length    = 1 longwords
                   DATA[00000000]: 00000009
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000000
          Mask length    = 8 bits
          Mask length    = 1 longwords
                   MASK[00000000]: FFFFFFFF
```

When this vector is activated, it initiates the pre-shift operation specified by macro number 2, which will send the TAP controller into the Capture-IR state. The TMS_SM line will then be driven to 0 (the loop bit in the macro) while the data in the vector is presented on the TDI_SM line for the next eight clock pulses. This data bit pattern is b00001001 or 0x09. This is the address of the SCANSTA111 multiplexer. The effect of executing this vector is to address the SCANSTA111 multiplexer.

The expected data is 0x00 but the mask is inverted. The actual mask value is 0x00. In the EVF2 record output produced by evf2dump, the mask should be interpreted as the inverse of the mask value in the SVF file. In other words, only bits which are 0 in the mask will be active in the data comparison. No data is compared at this step.

At the terminal count of the vector, meaning after 8 bits have been shifted in on the TDI_SM line, the TMS_SM line will be driven to 1. This puts the TAP controller in the Exit1-IR state. Then the macro instructs the SCANSTA101 STA master to output two more values on the TMS_SM line (the post-shift clock count is 2). These values are 1-0, which sends the TAP controller back to the Run-Test-Idle state.

The next record in the file is the register record that sets the START register to activate this vector. When this operation is complete, the SCANSTA111 multiplexer has been addressed and its TAP controller is in the Run-Test-Idle state.

Following is the next vector record:

```
===== VECTOR Sequence # 3 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number    = 2
          Macro number   = 2
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 3 longwords
          Data length    = 8 bits
          Data length    = 1 longwords
                   DATA[00000000]: 0000008E
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000025
          Mask length    = 8 bits
          Mask length    = 1 longwords
                   MASK[00000000]: FFFFFF00
```

When this vector is activated, it uses the same macro to load a new instruction into the SCANSTA111 multiplexer's instruction register. This instruction is op-code 0x8E, the MODESEL instruction. After this vector runs with macro 2, the TAP controller is once again in the Run-Test-Idle state. As before, the comparison mask is inverted so that only the 8 least-significant bits are used for the comparison.

Following a register record that activates this vector, another macro record is executed as seen below. The macro number 1 is loaded by this record.

```
===== MACRO Sequence # 3 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number   = 1
          Macro           = D2020320
```

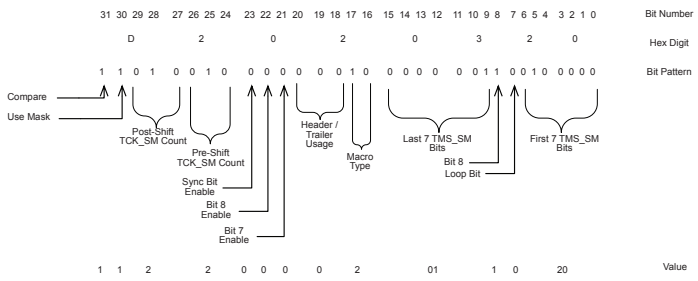

Figure 7-6. Bit Pattern of the Third Macro Record

This macro will have the effect of shifting data into the data register instead of the instruction register. Starting with the TAP controller in the Run-Test-Idle state, two bits are shifted out, 1-0. This sets the TAP controller to the Capture-DR state. The first bit is shifted into the data register in this state. The TAP controller

then enters the Shift-DR state, where it loops until the vector data has been shifted out. After the data from the vector has been shifted out, the TAP controller transitions to the Exit1-DR state, then to the Update-DR state, and then back to the Run-Test-Idle state.

Vector sequence number 4 uses this macro to shift the value 0x01 into the mode select 0 register of the SCANSTA111 multiplexer. This selects local scan port 0, as indicated in the SVF file.

Vector sequence number 5, shown below, uses macro number 2 to shift the UNPARK instruction into the instruction register of the SCANSTA111 multiplexer. As a reminder, the macro number 2 was previously set up to put the scan chain in the Capture-IR state, shift in an instruction, and leave the scan chain in the Run-Test-Idle state.

```
===== VECTOR Sequence # 5 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 8 ticks
          Slot number   = 2
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                    DATA[00000000]: 000000E7
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords
```

Following execution of this vector by register sequence number 11, which writes a value of 010 into the three LSBs of the START register, the scan chain consists of the SCANSTA111 multiplexer and the devices on local scan port 0. These devices are the SCAN921224 deserializer and the SCAN921023 serializer, in that order.

So now, when an instruction is shifted into the TAP, enough bits need to be shifted to fill the instruction registers for all three devices, plus a pad bit. The same macro, macro number 2, can be used to shift instructions into the instruction registers of all the devices just as it was used to shift instructions into the SCANSTA111 multiplexer.

Following is vector sequence number 6 which uses macro number 2 to shift the BYPASS instruction into the SCANSTA111 multiplexer and the SAMPLE instruction into the SCAN921224 deserializer and the SCAN921023 serializer. It should be noted that the least significant bit of the vector data is a pad bit.

```
===== VECTOR Sequence # 6 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 25 ticks
          Slot number   = 1
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 25 bits
          Data length   = 1 longwords
                    DATA[00000000]: 01FF0505
          Expect length = 25 bits
          Expect length = 1 longwords
                   EXPECT[00000000]: 00000202
          Mask length   = 25 bits
          Mask length   = 1 longwords
                    MASK[00000000]: FFFFF9F9
```

Now all the macros needed are in place. Macro number 4, which was the first one that was set up, issues a “five-high” TMS reset to the TAP controller. Macro number 3 places the TAP in the Run-Test-Idle state when it starts out in the Test-Logic-Reset state. Macro number 2 shifts data into the instruction register. It assumes that the TAP controller starts out in the Run-Test-Idle state and, when it is done, it leaves the TAP controller in the Run-Test-Idle state. Macro number 1 shifts data into the data register, also assuming that the TAP controller starts out in the Run-Test-Idle state and leaving it in this state when it is done.

If something else was needed besides this, new macros could be defined as needed, used, and then redefined. This isn’t necessary in this case, though. Vector sequence number 7, shown below, uses macro number 1 to shift the first test data into the boundary register.

```
===== VECTOR Sequence # 7 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number   = 3
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 6 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                    DATA[00000000]: FAAFFFFF
                    DATA[00000001]: 0000006D
          Expect length = 39 bits
          Expect length = 2 longwords
                   EXPECT[00000000]: 00000000
                   EXPECT[00000001]: 00000000
          Mask length   = 39 bits
          Mask length   = 2 longwords
                    MASK[00000000]: FFFFFFFF
                    MASK[00000001]: FFFFFFFF
```

The required 39 bits of data is too long to fit into a single 32-bit long word, so two long words are used. Also the mask, when inverted as described previously, specifies that no comparison

is to be done on the data shifted out of the TDO output of the scan chain.

As a reminder, the 39 bits of data are the bypass register bit in the SCANSTA111 multiplexer, the 19-bit boundary register of the SCAN921224 deserializer, the 18-bit boundary register of the SCAN921023 serializer, and a pad bit. After this data is loaded into the boundary registers of the SCAN921224 deserializer and SCAN921023 serializer, it will be presented at the output pins of these devices when the EXTEST instruction is issued.

Next, vector sequence number 8 issues the BYPASS instruction to the SCANSTA111 multiplexer (again), and the EXTEST instruction to the SCAN921224 deserializer and SCAN921023 serializer. It uses macro number 2 to write the data into the instruction registers of the parts.

```
===== VECTOR Sequence # 8 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 25 ticks
          Slot number   = 1
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 25 bits
          Data length   = 1 longwords
                    DATA[00000000]: 01FE0001
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords
```

Now the remainder of the file executes the same thing over and over. It scans data into the boundary registers of the devices on the board while scanning out and comparing the previous test results. The remainder of the file consists of vector sequences and register sequences such as these:

```
===== VECTOR Sequence # 9 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 6 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                    DATA[00000000]: FB3FFFFF
                    DATA[00000001]: 0000004E
          Expect length = 39 bits
          Expect length = 2 longwords
                  EXPECT[00000000]: E807FC96
                  EXPECT[00000001]: 00000000
          Mask length   = 39 bits
          Mask length   = 2 longwords
                    MASK[00000000]: 17F80069
                    MASK[00000001]: FFFFFFFF

===== REGISTER Sequence # 14 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011
```

Like the SVF file from which it was produced, this EVF2 file is for built-in self-test of a very simple scan chain with only three devices. For an operational board the scan chain might be much longer and the SVF and EVF2 files considerably more complex.

It is worth reiterating that the process that has just been carried out is not part of the normal flow for building a board built-in self-test. Normally it is neither necessary nor desirable (and maybe not even possible) to dissect an EVF2 file as has been done in this case. It is instructive, however, to perform this exercise at least once in order to gain an appreciation for what the test sequence is really accomplishing. In particular, examining the EVF2 file and the SVF file from which it was produced together illustrates how the commands in the SVF file are converted into register and memory reads and writes to the SCANSTA101 STA master.

The developer of a board built-in self-test does not need to go through this process, however. Almost always it is sufficient just to let the ATPG tool produce a SVF file and then convert it to an EVF2 file using EVF Workbench. The process shown here can be instructive for debugging if a self-test sequence is not working as expected, but normally this is not necessary.

8. Putting It All Together

In an effort to summarize, it is beneficial to consider step by step what has been implemented thus far, and how this could be applied to the development of a built-in self-test for a board.

First, a board designed to demonstrate the capabilities of National's JTAG support devices was utilized. The board was designed with JTAG-enabled devices. It was equipped with JTAG TAP connections on the backplane and through the connectors. And it was designed to accommodate the use of the SCANSTA111 Scan Bridge multiplexer.

From a hardware standpoint, this is really all that would be required of an operational system designed to support boundary scan built-in self-test. The JTAG support components could be "bolted on" to the board after the board interconnects were designed. The only additional interconnects required between the devices are those for the JTAG TAP itself.

Then the netlist for the board was taken, along with some additional information about the components on it and interconnects not shown on the netlist, and a built-in self-test sequence for the board was produced using a third-party ATPG tool. This process wasn't automatic, but it was relatively simple. The ATPG tool took most of the complexity out of the process of generating the built-in self-test. The output of the ATPG tool was a SVF file describing the test sequence in a human-readable form.

Next, the SVF file was taken and converted to an EVF2 file using National's EVF Workbench tool. This file encapsulated, in a binary format, the operations required to enable the SCANSTA101 STA master to perform the desired built-in self-test. The EVF2 file that results can be stored in some in-system memory and used for running the built-in self-test.

The interface function provided by National as C source code in the system controller software was included in order to deliver the instructions from the EVF2 file to the SCANSTA101 STA master. The interface function is flexible and easy to use. It ties into the remainder of the system controller software in a modular form that is easy to understand and implement.

These are the steps required to implement a built-in self-test for a board using the National's JTAG support devices. National supplies tools to convert the self-test sequence into the required format. National also supplies software source code for use in the embedded system software. This source code can be modified by the user if required. It can be used as-is in a wide variety of systems.

The design process described above is shown graphically in ***Figure 8-1***. The information and tools required to develop the built-in self-test are shown in the figure. The figure illustrates that the process of developing a board built-in self-test using JTAG is relatively simple and straightforward. All the steps in the development have been described in this design guide.

This design flow has been tested by working through all these steps for an example system, the SCANSTAEVK demonstration kit. In addition, for this simple system, the additional steps of examining the SVF and EVF2 files in detail were taken in order to understand what the test sequence is really accomplishing.

In the end, the self-test on the SCANSTAEVK demonstration kit was run. The intent was to demonstrate that when all the connections are good, the self-test passes, and when a connection is deliberately broken or a static logic level is deliberately set to an incorrect value, the self-test fails. This is what was observed, as described in the following section.

Putting It All Together

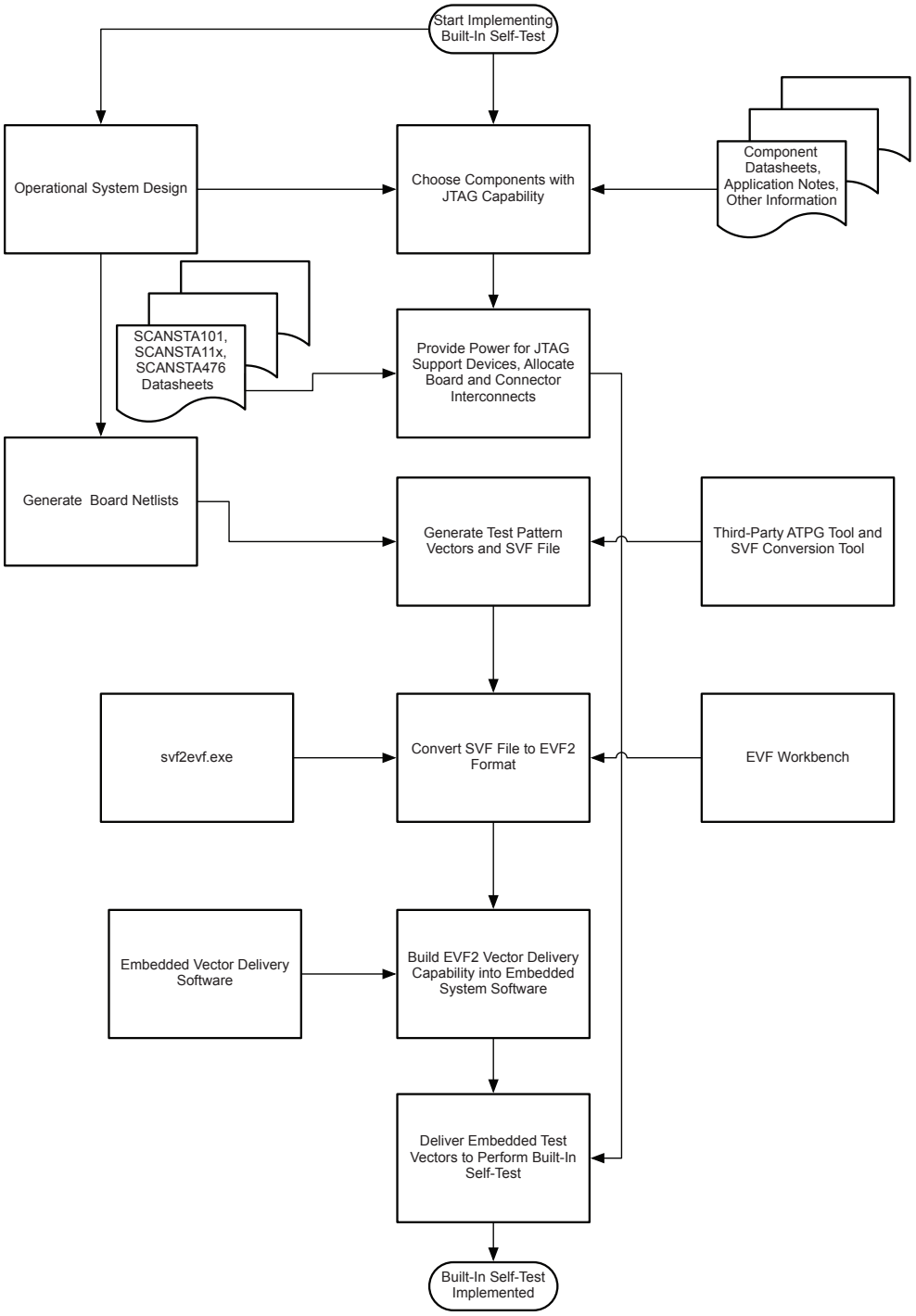

Figure 8-1. Design Flowchart for Built-In Self-Test

8.1. Demonstration of the Built-In Self-Test

As mentioned, the SCANSTAEVK demonstration kit includes software for delivering the test sequences to the board. This software uses an SCANSTA101 STA master implemented on a Corelis PCI-1149.1/101 JTAG controller. The PC-based software tool for JTAG vector delivery is called SCANEase.

The software uses the EVF2VectorDelivery interface described previously. In fact, the function prototypes used as examples of the required interface functions are taken from the SCANEase software. SCANEase includes a graphical user interface that enables the user to select an EVF2 file, or files, and to deliver these files to the target board.

After the EVF2 file was produced as described in previous sections, it was delivered to the SCANSTAEVK demonstration system by the SCANEase software. With all of the extra jumpers in place, and with a cable connecting connectors J4 and J5 shown in ***Figure 3-7***, the test ran to completion and passed. The output window of the SCANEase software is shown in ***Figure 8-2***.

It is comforting that the board self-test passes when there is nothing wrong with the board. But what if a fault is induced? The question is: Will the self-test detect it?

To answer that question, the cable from J4 was removed, breaking the connection between the differential output of the SCAN921023 serializer and the differential input of the SCAN921224 deserializer. In an operational system, if anything were to fail, it would likely be a cable, like this.

When the self-test was run in this condition, it did indeed fail, producing the output shown in ***Figure 8-3***. This indicates that the self-test detected the disconnection of the cable and indicated correctly that there was a problem on the board.

Introducing other faults on the board produced similar error indications at different steps in the test. For example, with reference to ***Figure 3-7***, the jumper J17 was removed to produce an error. Also, the position of switch S5 was changed which introduced a static 0 level on SCAN921023 serializer input DIN4. Each of these deliberately-induced board errors was detected. The switches in DIP switch S2 and S3 were also opened, producing errors when this changed the static input levels.

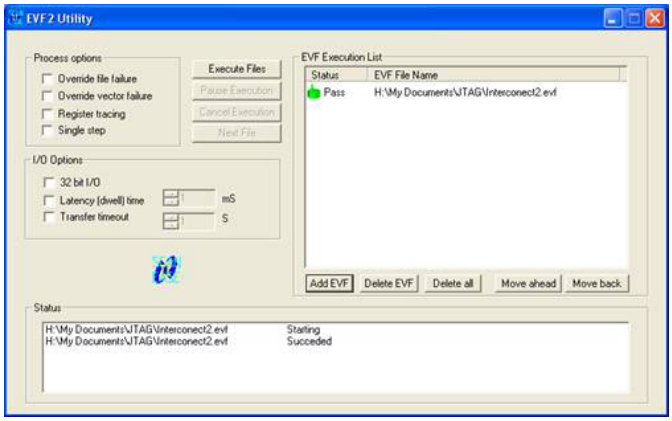

Figure 8-2. Screen Shot of the SCANEase Software indicating that the Board Self-Test Passed

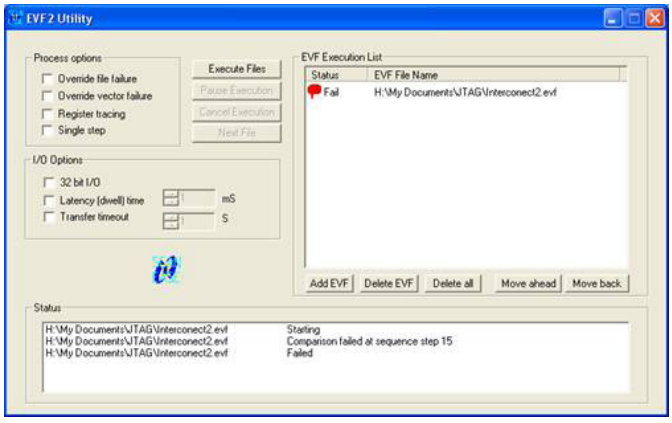

Figure 8-3. Screen Shot of the SCANEase Software indicating that the Board Self-Test Failed

In summary, all the faults introduced on the board were correctly detected by the self-test. This is exactly the desired behavior. This illustrates the utility of built-in self-test using boundary scan.

8.2. Summary and Conclusions

In conclusion, in the process of developing and deploying a board built-in self-test , it has been shown that it does what it was intended to do. Next, what it takes to develop and deploy such a test using the National's JTAG support devices is summarized.

When the design of the system begins, the designer doesn't need to think much about built-in self-test at all. Part of designing any system is choosing components that fulfill the requirements of the system. If the intention is to implement

Putting It All Together

a built-in self-test, the designer simply selects components that also implement boundary scan whenever possible. It is important to ensure there is enough board space and power supply capacity available for the JTAG support devices, such as the SCANSTA101 STA master and the SCANSTA111 multiplexer, that are to be used. And it is also critical to ensure that there are enough spare contacts on the connectors and enough spare traces on the backplane to accommodate the JTAG connections. If that has been done at the beginning of the system design, then it's very likely that the boundary scan-based built-in self-test in the system will be implemented successfully.

After the schematic for a board to be tested by boundary scan-based built-in self-test has been completed, the boundary scan support components and connections must be added. The boundary scan support devices are generally connected to the other components on the board only by the JTAG TAP.

The impact of boundary scan on board layout is minimal. Boundary scan generally does not require high-speed interconnects, so the TAP traces on the board can be routed without significant concern about signal integrity. The board traces related to the primary function of the board may need to be routed to support high-speed signals, but the TAP traces generally do not.

No matter what EDA tool was used to generate the board schematics, it is likely that the netlist output of that tool can be used for automated test pattern generation. Multiple vendors supply tools for automatic test pattern generation. All of them work off a netlist and a minimal amount of additional information. All the tools can produce boundary scan test sequences that can be deployed for built-in self-test. Any ATPG tool desired can be used to generate a SVF description of the boundary scan self-test to be performed.

The SVF file can be converted to an EVF2 format suitable for embedded test vector delivery by National's EVF Workbench tool. This tool works from a simple, intuitive GUI and produces EVF2 files that can be directly ported to storage on the system to be tested.

National provides source code to be added to the system software for delivering the test sequences to the SCANSTA101 STA master using a third-party ATPG tool. Incorporating this source code into the system software is straightforward for an experienced programmer.

Once all of these elements are in place, the system controller can initiate a built-in self-test that can diagnose many connection problems on the system. This process has been demonstrated on a model system. Also described in detail was what goes on "under the hood" of a boundary scan-based built-in self-test.

National's family of JTAG support devices provides a powerful capability for implementing boundary scan-based built-in self-test. The devices are supported by an extensive software suite provided both by ATPG software vendors and National. Adding built-in self-test to a board can reduce the overall cost of supporting the board. The minimal additional investment required to design built-in self-test into a system can provide payback many times over in improved reliability and lower maintenance and repair costs.

9.1. BSDL Files

The BSDL files for the SCANSTA111 JTAG multiplexer, SCAN921224 deserializer, and the SCAN921023 serializer follow in their entirety. These files describe, among other things, the op-codes associated with each instruction and the boundary register details for each device.

The BSDL file for the SCANSTA111 JTAG multiplexer follows:

```
-- SCANSTA111
--------------------------------------------------------------
-------------------
-- Copyright National Semiconductor Corporation 2000
--
-- STA111 ScanBridge 2
-- JTB   15 Sept 2000  Orignal
-- BillA 8 Oct 2000   Added S6 to data register
--
--
-- National Semiconductor Customer Service Center
-- N. America  (800) 272-9959
-- Europe Germany p49 (0) 69 9508 6208
--------------------------------------------------------------
-------------------
-- Revised Sept 12 02 to add BGA package and change ID code
version to 0001...

entity scansta111 is
  generic (PHYSICAL_PIN_MAP : string := "UNDEFINED");

  port (VCC: linkage bit_vector(2 downto 0);
        GND: linkage bit_vector(2 downto 0);
        TRSTB,TCKB,TMSB,TDIB: in bit;
        TDOB: out bit;
        TRISTB,YB,AB: linkage bit;
        OE: in bit;
        TEST_ENABLE: linkage bit;
      --  S6: linkage bit;
        S: in bit_vector(6 downto 0);
        TDI: linkage bit_vector(2 downto 0);
        TDO: linkage bit_vector(2 downto 0);
        TRIST: linkage bit_vector(2 downto 0);
        LSP_ACTIVE: linkage bit_vector(2 downto 0);
        TMS: linkage bit_vector(2 downto 0);
        TCK: linkage bit_vector(2 downto 0);
        TRST: linkage bit_vector(2 downto 0);
        Y: linkage bit_vector(1 downto 0);
        A: linkage bit_vector(1 downto 0)
        );

  use STD_1149_1_1990.all;   -- Get Std 1149.1-1990
attributes and definitions

  attribute PIN_MAP of scansta111 : entity is PHYSICAL_PIN_
MAP;

  constant TSSOP_PACKAGE:PIN_MAP_STRING:="VCC:(1,24,35),
GND:(25,36,48)," &
           "TRSTB:2, TCKB:3, TMSB:4, TDIB:5,TDOB:6,TRISTB:7,
YB:8, AB:9, OE:10," &
           "TEST_ENABLE:11,  S:(18, 17,16,15,14,13,12)," &
           "TDI:(19,33,44), TDO:(20,32,43), TRIST:(21,31,42),
LSP_ACTIVE:(22,28,39)," &
           "TMS:(23,34,45), TCK:(26,37,46), TRST:(27,38,47),
Y:(30,41), A:(29,40)";

  constant BGA_PACKAGE:PIN_MAP_STRING:="VCC:(A4,E4,E7),
GND:(C4,D5,G4)," &
           "TRSTB:B3, TCKB:A3, TMSB:B2,
TDIB:A2,TDOB:A1,TRISTB:C3, YB:C2, AB:B1, OE:D2," &
           "TEST_ENABLE:C1,  S:(G1,F1,F2,E1,E2,D1,D3)," &
           "TDI:(E3,F7,B5), TDO:(F3,E6,C5), TRIST:(G2,E5,A7),
LSP_ACTIVE:(F4,F6,C7)," &
           "TMS:(G3,D6,A6), TCK:(F5,D7,B4), TRST:(G5,C6,A5),
Y:(G7,B7), A:(G6,B6)";


  attribute TAP_SCAN_IN    of TDIB : signal is true;
  attribute TAP_SCAN_MODE  of TMSB : signal is true;
  attribute TAP_SCAN_OUT   of TDOB : signal is true;
  attribute TAP_SCAN_CLOCK of TCKB : signal is (25.0e6,
BOTH);
  attribute TAP_SCAN_RESET of TRSTB : signal is true;

  attribute INSTRUCTION_LENGTH of scansta111 : entity is 8;

  attribute INSTRUCTION_OPCODE of scansta111 : entity is
       "BYPASS     (11111111),"  &
       "EXTEST     (00000000),"  &
       "SAMPLE     (10000001),"  &
       "IDCODE     (10101010),"  &
       "UNPARK     (11100111),"  &
       "PARKTLR    (11000101),"  &
       "PARKRTI    (10000100),"  &
       "PARKPAUSE  (11000110),"  &
       "GOTOWAIT   (11000011),"  &
       "MODESEL    (10001110),"  &
       "MODESEL2   (10000011),"  &
       "MCGRSEL    (00000011),"  &
       "SOFTRESET  (10001000),"  &
       "LFSRSEL    (11001001),"  &
       "LFSRON     (00001100),"  &
       "LFSROFF    (10001101),"  &
       "CNTRSEL    (11001110),"  &
       "CNTRON     (00001111),"  &
       "CNTROFF    (10010000),"  &
       "TRANSPR0   (10100000),"  &
       "TRANSPR1   (10100001),"  &
       "TRANSPR2   (10100010),"  &
       "SGPIO0     (10111000),"  &
       "SGPIO1     (10111001),"  &
       "SGPIO2     (10111010)";

  attribute INSTRUCTION_CAPTURE of scansta111 : entity is
"XXXXXX01";

  attribute IDCODE_REGISTER of scansta111 : entity is
       "0001" &               -- Version
       "1111110000001111" &  -- Part number
       "00000001111" &        -- Manufacturer Identity
       "1";                    -- Manditory LSB

  attribute REGISTER_ACCESS of scansta111 : entity is
       "IDCODE (UNPARK,PARKTLR,PARKRTI,PARKPAUSE,GOTOWAIT,SO
FTRESET,LFSRON," &
            "LFSROFF,CNTRON,CNTROFF)," &
       "MODE[8] (MODESEL)," &
       "MODE2[8] (MODESEL2)," &
       "MCGR[2] (MCGRSEL)," &
       "LFSR[16] (LFSRSEL)," &
       "CNTR[32] (CNTRSEL)," &
       "TRANSPR0[8] (TRANSPR0)," &
       "TRANSPR1[8] (TRANSPR1)," &
       "TRANSPR2[8] (TRANSPR2)," &
       "SGPIO0[8] (SGPIO0)," &
       "SGPIO1[8] (SGPIO1)," &
       "SGPIO2[8] (SGPIO2)";

  attribute BOUNDARY_CELLS of scansta111 : entity is "BC_4";
  attribute BOUNDARY_LENGTH of scansta111 : entity is 8;

  attribute BOUNDARY_REGISTER of scansta111 : entity is
      -- num cell  port   function safe [ccell disval rslt]
        "0 (BC_4, S(0),   input,   X)," &              --
All inputs
```

Sample Files

```
        "1 (BC_4, S(1),   input,   X),"  &
        "2 (BC_4, S(2),   input,   X),"  &
        "3 (BC_4, S(3),   input,   X),"  &
        "4 (BC_4, S(4),   input,   X),"  &
        "5 (BC_4, S(5),   input,   X),"  &
        "6 (BC_4, S(6),   input,   X),"  &
        "7 (BC_4, OE,     input,   X)";

end scansta111;
```

The BSDL file for the SCAN921224 deserializer follows:

```
-- Copyright National Semiconductor Corporation 2001
--
-- Boundary Scan Description Language, BSDL Model for NSC_
SCAN921224
-- 10-bit LVDS Deserializer
--
-- National Semiconductor Customer Service Center
-- N. America  (800) 272-9959
-- Europe Germany p49 (0) 69 9508 6208
-------------------------------------------------------------
------------------
-- 01   Initial
-- 02   4 Feb 2001  Include relationship between
--                   cell_18 ctrl cell  and cell_05 RCLK
cell
-- 03   14 Mar 01   Verified through additional ATPG tools
--                   Changed BGA_49 to BGA_49_INTEGER. Added
BGA_49_BALL
--                   Add and commented out attribute PORT_
GROUPING for RIn
--                     as this attrubute is not handled by
all ATPG
--                   Corrected RUNBIST
--                   Corrected Boundary scan cell chain
length
--                   Corrected Control values for ROUT(7-9)
-- 04   21 Mar 01   Corrected ID
--                   Corrected cell ordering i.e. cell
closest TDO = 0
-- 05   29 Mar 01   Corrected control cells
-- 06   29 Apr 02   Corrected attribute ordering (RUNBIST_
EXECUTION)
-- 07   28 Aug 09   Uncommented RIn port and differential
port grouping.

entity NSC_SCAN921224 is
    generic (PHYSICAL_PIN_MAP : string := "BGA_49_BALL");

    port (
        ROUT:             out        bit_vector(0 to 9);
        RCLK_R_F:         in         bit;
        RIp:              in         bit;
        RIn:              in         bit;  -- 28 Aug 09
was commented out was linkage bit
        PWRDN:            in         bit;
        LOCK:             out        bit;
        RCLK:             out        bit;
        REN:              in         bit;
        REFCLK:           in         bit;
        TDI:              in         bit;
        TMS:              in         bit;
        TCK:              in         bit;
        TRST:             in         bit;
        TDO:              out        bit;
        DVCC:             linkage    bit_vector(4 downto
0);
        DGND:             linkage    bit_vector(6 downto
0);
        AVCC:             linkage    bit_vector(4 downto
0);
        AGND:             linkage    bit_vector(4 downto
0);
        NC:               linkage    bit_vector(3 downto
0)
        );

    use STD_1149_1_1994.all;

    attribute COMPONENT_CONFORMANCE of NSC_SCAN921224 :
entity is "STD_1149_1_1993";

    attribute PIN_MAP of NSC_SCAN921224 : entity is
PHYSICAL_PIN_MAP;

-- BGA_49_INTEGER identifies each pin as an integer
    constant BGA_49_INTEGER : PIN_MAP_STRING :=
        "ROUT:(18, 5, 11, 13, 21, 27, 42, 47, 40, 46)," &
        "RCLK_R_F:10," &
        "RIp:23," &
        "RIn:15," &      -- 28 Aug 09 was commented out
        "PWRDN:24," &
        "LOCK:29," &
        "RCLK:30,"&
        "REN:22,"&
        "REFCLK:3,"&
        "TDI:41,"&
        "TMS:49," &
        "TCK:33," &
        "TRST:34," &
        "TDO:48," &
        "DVCC:(7, 14, 19, 20, 26)," &
        "DGND:(1, 6, 12, 28, 32, 35, 45)," &
        "AVCC:(8, 16, 36, 37, 43)," &
        "AGND:(4, 9, 38, 39, 44)," &
        "NC:(2, 17, 25, 31)";

-- BGA_49_BALL identifies each pin by a "ball" identifier
    constant BGA_49_BALL : PIN_MAP_STRING :=
        "ROUT:(C4, A5, B4, B6, C7, D6, F7, G5, F5, G4)," &
        "RCLK_R_F:B3," &
        "RIp:D2," &
        "RIn:C1," &     -- 28 Aug 09 was commented out
        "PWRDN:D3," &
        "LOCK:E1," &
        "RCLK:E2,"&
        "REN:D1,"&
        "REFCLK:A3,"&
        "TDI:F6,"&
        "TMS:G7," &
        "TCK:E5," &
        "TRST:E6," &
        "TDO:G6," &
        "DVCC:(A7, B7, C5, C6, D5)," &
        "DGND:(A1, A6, B5, D7, E4, E7, G3)," &
        "AVCC:(B1, C2, F1, F2, G1)," &
        "AGND:(A4, B2, F3, F4, G2)," &
        "NC:(A2, C3, D4, E3)";

    attribute PORT_GROUPING of NSC_SCAN921224 : entity is
-- 28 Aug 09 was commented out
        "DIFFERENTIAL_VOLTAGE ( (RIp, RIn))";
-- 28 Aug 09 was commented out

    attribute TAP_SCAN_IN of TDI : signal is true;
    attribute TAP_SCAN_MODE of TMS : signal is true;
    attribute TAP_SCAN_OUT of TDO : signal is true;

    attribute TAP_SCAN_CLOCK of TCK : signal is (25.0e6,
BOTH);
    attribute TAP_SCAN_RESET of TRST : signal is true;

    attribute INSTRUCTION_LENGTH of NSC_SCAN921224 : entity
is 8;

    attribute INSTRUCTION_OPCODE of NSC_SCAN921224 : entity
is
        "BYPASS           (11111111)," &
```

```
        "EXTEST           (00000000)," &
        "SAMPLE           (10000010)," &
        "IDCODE           (10000001)," &
        "CLAMP            (10000111)," &
        "HIGHZ            (00000110)," &
        "RUNBIST          (10000011)";

    attribute INSTRUCTION_CAPTURE of NSC_SCAN921224 : entity
is "XXXXXX01";

    attribute IDCODE_REGISTER of NSC_SCAN921224 : entity is
        "1000" &                  -- version
        "1111110000100101" &     -- part number FC25 RX
        "00000001111" &           -- manufacturer's identity
        "1";                       -- required by 1149.1

    attribute REGISTER_ACCESS of NSC_SCAN921224 : entity is
        "BYPASS           (BYPASS, CLAMP, HIGHZ)," &
        "BOUNDARY         (SAMPLE, EXTEST)," &
        "BISTREG[2]       (RUNBIST)," &
        "DEVICE_ID        (IDCODE)";

--    attribute BOUNDARY_CELLS of NSC_SCAN921224 :entity is
"BC_1,BC_4";

    attribute BOUNDARY_LENGTH of NSC_SCAN921224 : entity is
19;

    attribute BOUNDARY_REGISTER  of NSC_SCAN921224 : entity
is
     --
     -- num   cell   port                    function   safe
[ccell disval rslt]
     --
        "18  (BC_1,  ROUT(7),          output3,  X,0, 0,
Z)," &
        "17  (BC_1,  ROUT(8),          output3,  X,0, 0,
Z)," &
        "16  (BC_1,  ROUT(9),          output3,  X,0, 0,
Z)," &
        "15  (BC_1,  *,                 controlr, 0)," &
        "14  (BC_1,  LOCK,             output3,  X, 15, 0,
Z)," &
        "13  (BC_1,  RCLK,             output3,  X,0, 0,
Z)," &
        "12  (BC_4,  REN,               input,    X)," &
        "11  (BC_4,  PWRDN,            input,    X)," &
        "10  (BC_4,  RIp,               input,    X)," &
        "9   (BC_4,  REFCLK,           input,    X)," &
        "8   (BC_4,  RCLK_R_F,         input,    X)," &
        "7   (BC_1,  ROUT(0),          output3,  X,0, 0,
Z)," &
        "6   (BC_1,  ROUT(1),          output3,  X,0, 0,
Z)," &
        "5   (BC_1,  ROUT(2),          output3,  X,0, 0,
Z)," &
        "4   (BC_1,  ROUT(3),          output3,  X,0, 0,
Z)," &
        "3   (BC_1,  ROUT(4),          output3,  X,0, 0,
Z)," &
        "2   (BC_1,  ROUT(5),          output3,  X,0, 0,
Z)," &
        "1   (BC_1,  ROUT(6),          output3,  X,0, 0,
Z)," &
        "0   (BC_1,  *,                 controlr, 0)";

    attribute RUNBIST_EXECUTION of NSC_SCAN921224 : entity
is
        "Wait_duration (10.0e-3), "&
        "Observing HIGHZ At_Pins, "&
        "Expect_Data 11";

end NSC_SCAN921224;
```

The BSDL file for the SCAN921023 serializer follows:

```
-------------------------------------------------------------
------------------
-- Copyright National Semiconductor Corporation 2001
--
-- Boundary Scan Description Language, BSDL Model for NSC_
SCAN921023
-- 10-bit LVDS Serializer
--
-- National Semiconductor Customer Service Center
-- N. America  (800) 272-9959
-- Europe Germany p49 (0) 69 9508 6208
-------------------------------------------------------------
------------------
-- 01   Initial
-- 02   14 Mar 01   Verified through additional ATPG tools
--                   Changed BGA_49 to BGA_49_INTEGER. Added
BGA_49_BALL
--                   Reversed order of DIN from (9 downto 0)
-> (0 to 9)
--                   Corrected ID code
--                   Corrected RUNBIST
-- 03   21 Mar 01   Corrected ID
--                   Corrected cell ordering i.e. cell
closest TDO = 0
-- 04   29 Mar 01   Corrected control cells
-- 05   29 Mar 01   Corrected disable value
-- 06   29 Apr 02   Corrected attribute ordering (RUNBIST_
EXECUTION) & fixed bist register name
-- 07   28 Aug 09   Uncommented DOn and differential port
grouping

entity NSC_SCAN921023 is
    generic (PHYSICAL_PIN_MAP : string := "BGA_49_BALL");

    port (

        DIN:              in          bit_vector(0 to 9);
        SYNC2:            in          bit;
        SYNC1:            in          bit;
        PWRDN:            in          bit;
        DOp:              out         bit;
        DOn:              out         bit;  -- 28 Aug 09
was commented out
        DEN:              in          bit;
        TCLK:             in          bit;
        TCLK_R_F:         in          bit;
        TDI:              in          bit;
        TMS:              in          bit;
        TCK:              in          bit;
        TRST:             in          bit;
        TDO:              out         bit;
        DVCC:             linkage     bit_vector(2 downto
0);
        DGND:             linkage     bit_vector(4 downto
0);
        AVCC:             linkage     bit_vector(4 downto
0);
        AGND:             linkage     bit_vector(4 downto 0)
        );


   use STD_1149_1_1994.all;

    attribute COMPONENT_CONFORMANCE of NSC_SCAN921023 :
entity is "STD_1149_1_1993";



    attribute PIN_MAP of NSC_SCAN921023 : entity is
PHYSICAL_PIN_MAP;

-- BGA_49_INTEGER identifies each pin as an integer
```

Sample Files

```
    constant BGA_49_INTEGER : PIN_MAP_STRING :=
         "DIN:(3, 8, 23, 15, 24, 22, 30, 29, 37, 39)," &
         "SYNC2:10," &
         "SYNC1:4," &
         "PWRDN:21," &
         "DOp:28," &
         "DOn:26," &     -- 28 Aug 09 was commented out
         "DEN:27,"&
         "TCLK:32,"&
         "TCLK_R_F:45,"&
         "TDI:36,"&
         "TMS:31," &
         "TCK:38," &
         "TRST:44," &
         "TDO:43," &
         "DVCC:(17, 18, 33)," &
         "DGND:(1, 16, 34, 40, 46)," &
         "AVCC:(5, 6, 11, 14 ,47)," &
         "AGND:(12, 13, 20, 35, 42)";

-- BGA_49_BALL identifies each pin by a "ball" identifier
    constant BGA_49_BALL : PIN_MAP_STRING :=
         "DIN:(A3,B1, D2, C1, D3, D1, E2, E1, F2, F4)," &
         "SYNC2:B3," &
         "SYNC1:A4," &
         "PWRDN:C7," &
         "DOp:D7," &
         "DOn:D5," &    -- 28 Aug 09 was commented out
         "DEN:D6,"&
         "TCLK:E4,"&
         "TCLK_R_F:G3,"&
         "TDI:F1,"&
         "TMS:E3," &
         "TCK:F3," &
         "TRST:G2," &
         "TDO:G1," &
         "DVCC:(C3, C4, E5)," &
         "DGND:(A1, C2, E6, F5, G4)," &
         "AVCC:(A5, A6, B4, B7 ,G5)," &
         "AGND:(B5, B6, C6, E7, F7)";

    attribute PORT_GROUPING of NSC_SCAN921023 : entity is
-- 28 Aug 09 was commented out
         "DIFFERENTIAL_VOLTAGE ( (DOp, DOn))";
-- 28 Aug 09 was commented out

    attribute TAP_SCAN_IN of TDI : signal is true;
    attribute TAP_SCAN_MODE of TMS : signal is true;
    attribute TAP_SCAN_OUT of TDO : signal is true;

    attribute TAP_SCAN_CLOCK of TCK : signal is (25.0e6,
BOTH);
    attribute TAP_SCAN_RESET of TRST : signal is true;

    attribute INSTRUCTION_LENGTH of NSC_SCAN921023 : entity
is 8;

    attribute INSTRUCTION_OPCODE of NSC_SCAN921023 : entity
is
         "BYPASS           (11111111)," &
         "EXTEST           (00000000)," &
         "SAMPLE           (10000010)," &
         "IDCODE           (10000001)," &
         "CLAMP            (10000111)," &
         "HIGHZ            (00000110)," &
         "RUNBIST          (10000011)";

    attribute INSTRUCTION_CAPTURE of NSC_SCAN921023 : entity
is "XXXXXX01";

    attribute IDCODE_REGISTER of NSC_SCAN921023 : entity is
         "1000" &                  -- version
         "1111110000100110" &     -- part number FC26 TX
         "00000001111" &          -- manufacturer's identity
         "1";                      -- required by 1149.1

    attribute REGISTER_ACCESS of NSC_SCAN921023 : entity is
         "BYPASS           (BYPASS, CLAMP, HIGHZ)," &
         "BOUNDARY         (SAMPLE, EXTEST)," &
         "BISTREG[2]       (RUNBIST)," &
         "DEVICE_ID        (IDCODE)";

--    attribute BOUNDARY_CELLS of NSC_SCAN921023 :entity is
"BC_1,BC_4";

       attribute BOUNDARY_LENGTH of NSC_SCAN921023 : entity
is 18;

    attribute BOUNDARY_REGISTER  of NSC_SCAN921023 : entity
is
      --
      -- num   cell   port                    function   safe
[ccell disval rslt]
      --
         "17  (BC_4,  DIN(8),            input, X)," &
         "16  (BC_4,  DIN(7),            input, X)," &
         "15  (BC_4,  DIN(6),            input, X)," &
         "14  (BC_4,  DIN(5),            input, X)," &
         "13  (BC_4,  DIN(4),            input, X)," &
         "12  (BC_4,  DIN(3),            input, X)," &
         "11  (BC_4,  DIN(2),            input, X)," &
         "10  (BC_4,  DIN(1),            input, X)," &
         "9   (BC_4,  DIN(0),            input, X)," &
         "8   (BC_4,  SYNC2,             input, X)," &
         "7   (BC_4,  SYNC1,             input, X)," &
         "6   (BC_4,  PWRDN,             input, X)," &
         "5   (BC_1,  DOp,               output3,       X,
4, 0, Z)," &
         "4   (BC_1,  *,                 controlr,      0),"
&
         "3   (BC_4,  DEN,               input, X)," &
         "2   (BC_4,  TCLK,              input, X)," &
         "1   (BC_4,  TCLK_R_F,          input, X)," &
         "0   (BC_4,  DIN(9),            input, X)";

   attribute RUNBIST_EXECUTION of NSC_SCAN921023 : entity is
      "Wait_Duration (10.0e-3), "&
      "Observing HIGHZ At_Pins, "&
      "Expect_Data 01";

end NSC_SCAN921023;
```

9.2. SVF file

The SVF file produced for the self-test described in this design guide follows in its entirety.

```
!-------------------------------------------------------------
----------------
!
! Generated by Corelis CVFtoSVF Converter Version 1.07
!
! Generated from C:\Documents and Settings\CJRJSC\My
Documents\Scan\ScanExpress_Projects\SCANSTAEVK_Demo\JTAG_EVK_
Demo_interconnect_ic.cvf
! CVF File Version: 1.01
! CVF Test Name    : JTAG_EVK
! CVF Test Type    : INTERCON
! CVF Revision     : 1.01
! CVF File Date    : 091009
!
!-------------------------------------------------------------
----------------
!
!            SVF FILE STATEMENTS OP-CODE SYNTAX SUMMARY
!            --- ---- ---------- ------ ------ -------
!
! ENDDR   - Specify the end state for any data register
```

```
(SDR) scan operation
!
! ENDIR   - Specify the end state for any instruction
register (SIR) scan
!            operation
!
! RUNTEST - Forces the IEEE 1149.1 bus to the specified run
state for
!            a specified number of clocks. This can be used
to control RUNBIST
!            operation in the target
!
! SDR      - Performs an IEEE 1149.1 data register scan
!            Shift data op-code, followed by number-of-bits
(decimal),
!            serial-data-out (hex), expected-serial-data-in
(hex), mask-of-
!            serial-data-in (hex)
!
! SIR      - Performs an IEEE 1149.1 instruction register
scan
!            Shift instruction op-code, followed by number-
of-bits (decimal),
!            serial-data-out (hex), expected-serial-data-in
(hex), mask-of-
!            serial-data-in (hex)
!
! STATE    - Move the boundary scan controller state-machine
to this stable
!            state
!
! TRST     - Controls the optional Test Reset line
!
!-------------------------------------------------------------
----------------
!
!            NOTE: Text comments are marked with a preceding
'!' character
!            NOTE: MASK bit value of '0' masks out the
relevant TDO bit
!            NOTE: Serial-data-out (TDI) is shifted out with
LSB first
!
!-------------------------------------------------------------
----------------

TRST OFF;

STATE IDLE;

!-------------------------------------------------------------
----------------
!
! Source CVF file generated by ScanPlus TPG Version 2.03
!
!-------------------------------------------------------------
----------------

SIR         8  TDI (09)
               TDO (00)
              MASK (00);

SIR         8  TDI (8E)
               TDO (25)
              MASK (FF);

SDR         8  TDI (01)
               TDO (00)
              MASK (00);

SIR         8  TDI (E7)
               TDO (25)
              MASK (FF);

!-------------------------------------------------------------
----------------
!
! TOPOLOGY INFORMATION
! The devices are listed in the order from TDI to TDO of the
board
!
!   STA_U1       (SCANSTA111 Enhanced Scan Bridge at address
0x00000009 on TAP0)
!                          - Instruction Length ( 8)
Boundary Length (  1)
!   U1           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 19)
!   U2           (on TAP0) - Instruction Length ( 8) Boundary
Length ( 18)
!   PAD          (on TAP0) - Instruction Length ( 1) Boundary
Length (  1)
!
!-------------------------------------------------------------
----------------

SIR        25  TDI (1FF0505)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (6DFAAFFFFF)
               TDO (0000000000)
              MASK (0000000000);

SIR        25  TDI (1FE0001)
               TDO (0000202)
              MASK (0000606);

SDR        39  TDI (4EFB3FFFFF)
               TDO (00E807FC96)
              MASK (00E807FF96);

SDR        39  TDI (77FC3FFFFF)
               TDO (00E807FE96)
              MASK (00E807FF96);

SDR        39  TDI (47FFCFFFFF)
               TDO (00E807FE96)
              MASK (00E807FF96);

SDR        39  TDI (7FFFFFFFFF)
               TDO (00E807FE96)
              MASK (00E807FF96);

SDR        39  TDI (56FD5FFFBF)
               TDO (00E807FE96)
              MASK (00E807FF96);

SDR        39  TDI (75FCCFFFBF)
               TDO (00C807FE96)
              MASK (00E807FF96);

SDR        39  TDI (4CFBCFFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (7CF83FFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (44F80FFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (67F99FFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (7DFF0FFFBF)
               TDO (00C807FE96)
              MASK (00E807FF96);
```

```
SDR        39  TDI (74FBFFFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (7CF83FFFBF)
               TDO (00C807FC96)
              MASK (00E807FF96);

SDR        39  TDI (7BFFF7FFDF)
               TDO (00C807FC96)
              MASK (00E807FF96);

! Total number of vectors : 21
```

9.3. EVF2 File

The EVF2 file, converted to human-readable format by Evf2Dump, follows:

```
Verbose mode
=============== HEADER ====================
Largest vector bit length: 39 bits
Largest buffer bit length: 0 bits
Macro record count: 4
Vector record count: 23
Buffer record count: 0
Register record count: 28
Assumed initial state: Undefined
Defined final state: Undefined
==========================================
===== REGISTER Sequence # 0 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000004
          Data           = 00000004
          Register bits = ............................1..

===== REGISTER Sequence # 1 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 5 (CLKDIV)
          Mask           = 000000FE
          Data           = 00000020
          Register bits = .......................0010000.

===== REGISTER Sequence # 2 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000003
          Data           = 00000040
          Register bits = ..............................00

===== REGISTER Sequence # 3 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 2 (INTCTRL)
          Mask           = 00001E00
          Data           = 00000000
          Register bits = ...................0000.........

===== MACRO Sequence # 0 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 4
          Macro          = 0503007C

===== VECTOR Sequence # 0 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number    = 0
          Macro number   = 4
          Load on fly    = NO
          Compare data   = NO
          Use mask       = NO
          Array Length   = 0 longwords
          Data length    = 0 bits
          Data length    = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 4 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = ..............................001

===== REGISTER Sequence # 5 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 4 (SETUPR)
          Mask           = 00000008
          Data           = 00000043
          Register bits = ............................0...

===== MACRO Sequence # 1 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 3
          Macro          = 01030000

===== VECTOR Sequence # 1 =====
          Fixed Length = 32 bytes
          Total Length  = 32 bytes
          Clocks         = 0 ticks
          Slot number    = 0
          Macro number   = 3
          Load on fly    = NO
          Compare data   = NO
          Use mask       = NO
          Array Length   = 0 longwords
          Data length    = 0 bits
          Data length    = 0 longwords
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 6 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000001
          Register bits = ..............................001

===== MACRO Sequence # 2 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 2
          Macro          = D3020330

===== VECTOR Sequence # 2 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number    = 3
          Macro number   = 2
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
```

```
          Array Length  = 3 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                   DATA[00000000]: 00000009
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000000
          Mask length   = 8 bits
          Mask length   = 1 longwords
                   MASK[00000000]: FFFFFFFF

===== REGISTER Sequence # 7 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 3 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 8 ticks
          Slot number   = 2
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                   DATA[00000000]: 0000008E
          Expect length = 8 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000025
          Mask length   = 8 bits
          Mask length   = 1 longwords
                   MASK[00000000]: FFFFFF00

===== REGISTER Sequence # 8 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== MACRO Sequence # 3 =====
          Fixed Length = 20 bytes
          Total Length  = 20 bytes
          Macro number  = 1
          Macro          = D2020320

===== VECTOR Sequence # 4 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 8 ticks
          Slot number   = 3
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                   DATA[00000000]: 00000001
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 9 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 5 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 8 ticks
          Slot number   = 2
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 1 longwords
          Data length   = 8 bits
          Data length   = 1 longwords
                   DATA[00000000]: 000000E7
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 10 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 6 =====
          Fixed Length = 32 bytes
          Total Length  = 44 bytes
          Clocks         = 25 ticks
          Slot number   = 1
          Macro number  = 2
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 3 longwords
          Data length   = 25 bits
          Data length   = 1 longwords
                   DATA[00000000]: 01FF0505
          Expect length = 25 bits
          Expect length = 1 longwords
                 EXPECT[00000000]: 00000202
          Mask length   = 25 bits
          Mask length   = 1 longwords
                   MASK[00000000]: FFFFF9F9

===== REGISTER Sequence # 11 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000002
          Register bits = ............................010

===== VECTOR Sequence # 7 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number   = 3
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask       = YES
          Array Length  = 6 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FAAFFFFF
                   DATA[00000001]: 0000006D
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: 00000000
```

Sample Files

```
                EXPECT[00000001]: 00000000
          Mask length    = 39 bits
          Mask length    = 2 longwords
                   MASK[00000000]: FFFFFFFF
                   MASK[00000001]: FFFFFFFF

===== REGISTER Sequence # 12 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000004
          Register bits = ............................100

===== VECTOR Sequence # 8 =====
          Fixed Length = 32 bytes
          Total Length  = 36 bytes
          Clocks         = 25 ticks
          Slot number    = 1
          Macro number   = 2
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 1 longwords
          Data length    = 25 bits
          Data length    = 1 longwords
                   DATA[00000000]: 01FE0001
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 13 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000002
          Register bits = ............................010

===== VECTOR Sequence # 9 =====
          Fixed Length = 32 bytes
          Total Length  = 56 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 6 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FB3FFFFF
                   DATA[00000001]: 0000004E
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: E807FC96
                 EXPECT[00000001]: 00000000
          Mask length    = 39 bits
          Mask length    = 2 longwords
                   MASK[00000000]: 17F80069
                   MASK[00000001]: FFFFFFFF

===== REGISTER Sequence # 14 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 10 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 4 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FC3FFFFF
                   DATA[00000001]: 00000077
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: E807FE96
                 EXPECT[00000001]: 00000000
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 15 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 11 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 2 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FFCFFFFF
                   DATA[00000001]: 00000047
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 16 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 12 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 2 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FFFFFFFF
                   DATA[00000001]: 0000007F
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 17 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
```

```
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 13 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 2 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FD5FFFBF
                   DATA[00000001]: 00000056
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 18 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 14 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 4 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FCCFFFBF
                   DATA[00000001]: 00000075
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: C807FE96
                 EXPECT[00000001]: 00000000
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 19 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 15 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 4 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: FBCFFFBF
                   DATA[00000001]: 0000004C
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: C807FC96
                 EXPECT[00000001]: 00000000
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 20 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 16 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 2 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: F83FFFBF
                   DATA[00000001]: 0000007C
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 21 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 17 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
          Compare data   = YES
          Use mask       = YES
          Array Length   = 2 longwords
          Data length    = 39 bits
          Data length    = 2 longwords
                   DATA[00000000]: F80FFFBF
                   DATA[00000001]: 00000044
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length    = 0 bits
          Mask length    = 0 longwords

===== REGISTER Sequence # 22 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id    = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ............................011

===== VECTOR Sequence # 18 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number    = 2
          Macro number   = 1
          Load on fly    = NO
```

Sample Files

```
          Compare data  = YES
          Use mask      = YES
          Array Length  = 2 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: F99FFFBF
                   DATA[00000001]: 00000067
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 23 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011

===== VECTOR Sequence # 19 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 4 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FF0FFFBF
                   DATA[00000001]: 0000007D
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: C807FE96
                 EXPECT[00000001]: 00000000
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 24 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011

===== VECTOR Sequence # 20 =====
          Fixed Length = 32 bytes
          Total Length  = 48 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 4 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FBFFFFBF
                   DATA[00000001]: 00000074
          Expect length = 39 bits
          Expect length = 2 longwords
                 EXPECT[00000000]: C807FC96
                 EXPECT[00000001]: 00000000
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 25 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011

===== VECTOR Sequence # 21 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 2 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: F83FFFBF
                   DATA[00000001]: 0000007C
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 26 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011

===== VECTOR Sequence # 22 =====
          Fixed Length = 32 bytes
          Total Length  = 40 bytes
          Clocks         = 39 ticks
          Slot number   = 2
          Macro number  = 1
          Load on fly   = NO
          Compare data  = YES
          Use mask      = YES
          Array Length  = 2 longwords
          Data length   = 39 bits
          Data length   = 2 longwords
                   DATA[00000000]: FFF7FFDF
                   DATA[00000001]: 0000007B
          Expect length = 0 bits
          Expect length = 0 longwords
          Mask length   = 0 bits
          Mask length   = 0 longwords

===== REGISTER Sequence # 27 =====
          Fixed Length = 24 bytes
          Total Length  = 24 bytes
          Register id   = 0 (START)
          Mask           = 00000007
          Data           = 00000003
          Register bits = ..............................011

DUMP COMPLETE
```

10. References

1. "IEEE Standard Test Access Port and Boundary-Scan Architecture", *IEEE Standard 1149.1-2001*. New York, NY: IEEE Standards Board, 2001.
2. "IEEE Standard for In-System Configuration of Programmable Devices", *IEEE Standard 1532-2002*. New York, NY: IEEE Standards Board, 2003.
3. National Semiconductor Corp., SCANSTA101 Low-Voltage IEEE 1149.1 STA Master datasheet, 2006.
4. National Semiconductor Corp., SCANSTA111 Enhanced Scan Bridge Multidrop Addressable IEEE 1149.1 (JTAG) Port datasheet, 2005.
5. National Semiconductor Corp., SCANSTA112 7-Port Multidrop IEEE 1149.1 (JTAG) Multiplexer datasheet, 2005.
6. National Semiconductor Corp., SCANSTA476 Eight-Input IEEE 1149.1 Analog Voltage Monitor datasheet, 2005.
7. K. P. Parker, *The Boundary Scan Handbook*, Third Edition. Norwell, MA: Kluwer Academic Publishers, 2003.
8. National Semiconductor Corp., SCAN921023 and SCAN921224 20 MHz to 60 MHz 10-Bit Bus LVDS Serializer and Deserializer with IEEE 1149.1 (JTAG) and At-Speed BIST datasheet, 2004.
9. National Semiconductor Corp., Application Note AN-1259, SCANSTA112 Designer's Reference, 2009.
10. JTAG Technologies, "Design for Test Guidelines For Board Testing and In-System Configuration". JTAG Technologies, June, 2008.
11. JTAG Technologies, "Design for Test Guidelines For System-Level Testing and In-System Configuration". JTAG Technologies, June 2008.
12. "IEEE Standard for Boundary-Scan Testing of Advanced Digital Networks", *IEEE Standard 1149.6-2003*. New York, NY: IEEE Standards Board, 2003.
13. P. J. Ashenden, *The Designer's Guide to VHDL*, Third Edition. Burlington, MA: Elsevier, Inc., 2008.
14. K. L. Short, *VHDL for Engineers*. Upper Saddle River, New Jersey: Pearson Education, Inc., 2009.
15. N. Jarwala and C. W. Yau, "A Unified Theory for Designing Optimal Test Generation and Diagnosis Algorithms for Board Interconnects", in *Proceedings, International Test Conference*, 1989, pp. 71-77.
16. C. W. Yau and N. Jarwala, "A New Framework for Analyzing Test Generation and Diagnosis Algorithms for Wiring Interconnects", in *Proceedings, International Test Conference*, 1989, pp. 63-70.
17. W. H. Kautz, "Testing for Faults in Wiring Networks", *IEEE Transactions on Computers*, vol. C-23, No. 4, April, 1974, pp. 358-363.
18. Asset Intertech, Inc., *Serial Vector Format Specification*, Revision E, 1999.

Worldwide Design Centers and Manufacturing Facilities

Design Centers

USA:
Chandler, Arizona
Federal Way, Washington
Fort Collins, Colorado
Grass Valley, California
Indianapolis, Indiana
Longmont, Colorado
Norcross, Georgia
Phoenix, Arizona
Salem, New Hampshire
Santa Clara, California
South Portland, Maine
Tucson, Arizona

EUROPE:
Delft, Netherlands
Unterhaching, Germany
Greenock, Scotland
Milan, Italy
Oulu, Finland
Tallinn, Estonia

ASIA:
Bangalore, India
Hangzhou, China
(joint with Zhejiang University)
Hong Kong, China
Tokyo, Japan

Manufacturing Facilities

Wafer (Die) Fabrication:
Arlington, Texas
South Portland, Maine
Greenock, Scotland

Chip Test and Assembly:
Melaka, Malaysia

World Headquarters
2900 Semiconductor Drive
Santa Clara, CA 95051
USA
+1 408 721 5000
www.national.com

Mailing Address:
PO Box 58090
Santa Clara, CA 95052
support@nsc.com

European Headquarters
Livry-Gargan-Str. 10
82256 Fürstenfeldbruck
Germany
+49 8141 35 0
europe.support@nsc.com

Asia Pacific Headquarters
2501 Miramar Tower
1 Kimberley Road
Tsimshatsui, Kowloon
Hong Kong
+852 2737 1800
ap.support@nsc.com

Japan Headquarters
Beside KIBA
2-17-16
Kiba, Koto-ku
Tokyo, 135-0042, Japan
+81 3 5639 7300
jpn.feedback@nsc.com

National Semiconductor