Programming Tools User Guide for Radiant Software

November 7, 2019

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

Programming Tools Description

This user guide is intended to provide users with basic information, and references on where to find more detailed information, to assist in configuring and programming Lattice devices using Radiant software Programmer and related tools including Deployment Tool, Programming File Utility, and Download Debugger.

Programmer

Radiant software Programmer is a system for programming devices. The software supports serial programming of Lattice devices using PC and Linux environments. The tool also supports embedded microprocessor programming. Refer to “Programmer Overview” on page 11.

Deployment Tool

The Deployment Tool graphical user interface (GUI) is separate from the Radiant and Programmer design environment, and is launched from Radiant Programmer by choosing **Tools > Deployment Tool**.

The Deployment Tool allows you to generate files for deployment for single devices, a chain of devices, and can also convert data files to other formats and use the data files it produces to generate other data file formats. Refer to “Deployment Tool Overview” on page 19.

Programming File Utility

Programming File Utility allows the viewing and editing of different format programming files.The Programming File Utility is a stand-alone tool that allows you to view and compare data files. When comparing two data files, the software generates an output (.out) file with the differences highlighted in red.

Detailed information and procedures on how to use the Programming File Utility are contained in the “Using Programming File Utility” section of the Lattice Radiant online help.

Download Debugger

Download Debugger is a utility for debugging programming for debugging Serial Vector Format (SVF) files, Standard Test And Programming Language (STAPL) files, and Lattice Embedded (VME) files. Download Debugger allows you to program a device, and edit, debug, and trace the process of SVF, STAPL, and VME files. Download Debugger also allows you to create, edit, or view a VME file in hexadecimal format.

Detailed information and procedures on how to use the Download Debugger are contained in the “Debugging SVF, STAPL, and VME Files” section of the Lattice Radiant software online help or in the stand-alone Download Debugger online help.

Embedded Flow

Programming flow using a processor to read the contents of a stored programming file and programming the FPGA. Lattice provides the option to generate several different file formats for different embedded target options. Refer to “Embedded Flow Overview” on page 27.

Driver Installation

A utility is available for installing programming drivers after the Radiant software or Programmer software has been installed. Refer to the topic “Installing/Uninstalling Parallel Port Driver and USB Driver on a PC” in the Lattice Radiant online help or in the Programmer online help.

Programmer Overview

Usage and flow

Programming is the process changing the state of a non-volatile programmable element (such as embedded Flash and external SPI Flash devices) by downloading data from bitstream or hex files transmitted to the download cable through the host computer's serial communications port; from an embedded system; or from a third-party programmer.

Configuring is the process of changing the state of a volatile programmable element (such as SRAM in the FPGA).

Before programming or configuring an FPGA, you need to create and verify your design, and then generate data files. To download a data file into the target device, use the Programmer tool.

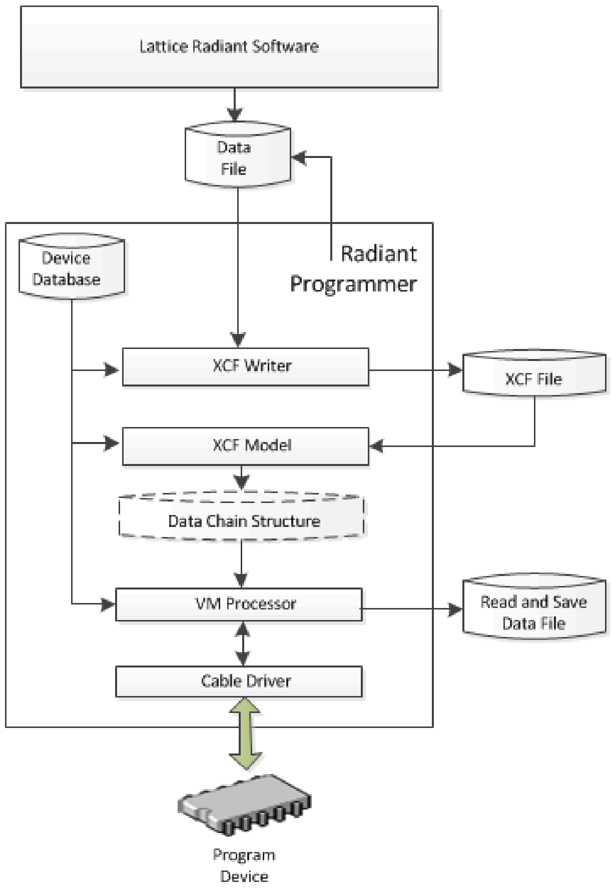
Programmer Design Flow

The design flow for creating and downloading a design chain is the same, whether you select the devices and settings from the software or use an existing chain configuration file.

1. Create a design and compile it to a ISC, hex, or bitstream data file.
2. Using Programmer, create a new chain configuration or open an existing one.
3. Add devices to the chain, and select the data file and operation for each.
4. Arrange the order of the devices in the chain and edit the options for each device as needed.
5. Program the daisy-chained devices using the Program toolbar command.

Figure 1 on page 12 describes the Lattice programming process.

Figure 1: Programming Design Flow



Programming Basics

To successfully program devices in-system, there are a few simple requirements that must first be met. The first of these requirements is that the devices on the board must be correctly connected into an 1149.1 scan chain. This scan chain can be used for either programming or testing the board.

To program using Programmer a description of the scan chain must be developed. This description, called a chain file, contains basic information about all of the devices in the chain. For the Lattice devices, it includes the device type, the operation to be performed, and the data file, if required by the operation. Additional information in the chain file can include the state of the I/O pins during programming, along with security requirements. If the chain includes non-Lattice devices, the instruction register length is required for these devices. The instruction register length can be found from the BSDL file or the SVF file for the device.

Another requirement for successful programming is thoughtful board design. The signals used in a scan chain (TCK, TMS, TDI, and TDO) rarely operate as fast as the data path signals on the board. However, correct board layout methodologies, such as buffering for large chains and termination resistors, are required to ensure trouble-free operation. Some Lattice devices have some additional pins (TRST, ispEN, PROGRAMN, INITN, DONE, SLEEPN, and TOE) that can affect boundary scan programming and test if not taken care of properly.

After all of these requirements have been met, it should be relatively straightforward to program any number of devices on a board. You can program the devices using a PC or Linux system and a board test system connected by one of the following cables:

- A Lattice parallel port cable with the 8-pin AMP connector or 10-pin JEDEC connector
- A Lattice USB port cable

In-System Programming

After you have compiled your design to a data file (hex or bitstream) and device programming is necessary, you must serially shift the fuse map (a fuse map file is a design file that has the fuse data already pre-arranged in exactly the same format as the physical layout of the fuse array of the device) data into the device along with the appropriate addresses and commands.

Lattice non-volatile FPGA devices use embedded flash memory and require only TTL-level programming signals. An integrated state machine controls the sequence of programming operations, such as identifying the device, shifting in the appropriate data and commands, and controlling internal signals to program and erase the embedded Flash in the device. Programming consists of serially shifting the logic implementation stored in a data file into the device along with appropriate addresses and commands, programming the data into the embedded Flash, and shifting the data from the logic array out for device programming verification.

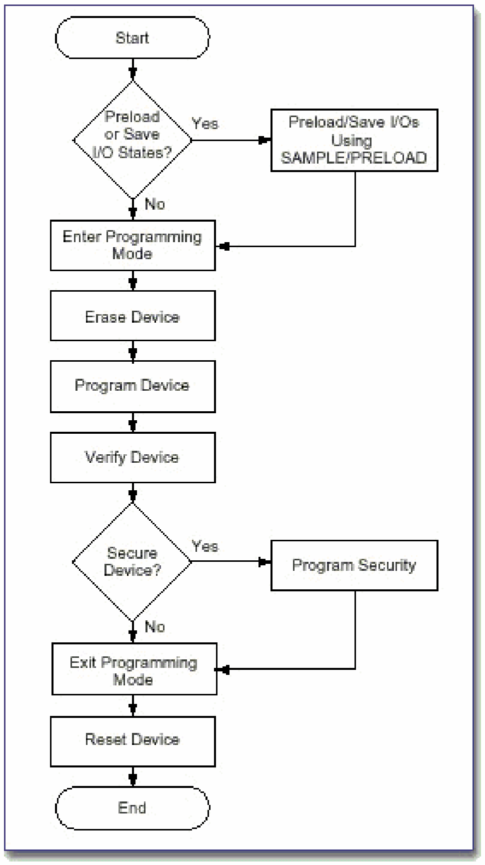
Most of Lattice's devices use the IEEE 1149.1-1993 Boundary Scan Test Access Port (TAP) as the primary interface for in-system programming.

Programming Algorithm Basics

Programming a device is similar to programming any piece of memory, such as an EPROM or a Flash memory. The device can be thought of as an array that is programmed one row at a time. The programming information is provided to the software in the form of a data file that must be converted into the row and column fuse map data. Before a non-volatile device can be programmed, it first has to be erased. After the device has been erased, the programming data can be loaded and the device programmed. After the device has been programmed, it will be verified by reading out the data in the device and comparing it to the original.

Figure 2 on page 14 shows the basic programming flow for the device. It does not include the data file conversion into fuse map data, for it assumes that step has already been done. This programming flow will be the same, regardless of the programming hardware used.

Figure 2: Basic Device Programming Flow



Note

If the device will not be programmed in-circuit (that is, by a cable or an embedded processor), it is not necessary to preload or save the I/O states.

Programming Times

The time it takes to program a device can often determine where in the manufacturing process a device, or group of devices, is programmed. A board test system costing hundreds of thousands of dollars to purchase and as much as one dollar per minute to operate can be an expensive alternative to programming if programming times are too long. In many instances, it is more cost-effective to buy a couple of PCs and program the devices using these much cheaper systems.

The time it takes to completely program a device is based on the time it takes to first erase the device, then to program each row in the device, and finally to verify the device. The erase time for all devices is between 100 ms and 200 ms. A single row is programmed in 10 ms to 50 ms, depending on the device. The verification process is the quickest of the required steps in the programming sequence and mainly depends on the time required to shift the verification data out of any given device.

The benefit of minimal programming times is much more obvious on board test systems, because they are included as a part of the test program and are running at the fastest speed possible. Additionally, there is no translation needed to or from the data file, since this has already been done by Programmer.

USERCODE

User-programmable identification can ease problems associated with document control and device traceability. Every Lattice 1149.1-compliant device contains a 32-bit register that is accessible through the optional IEEE 1149.1 USERCODE instruction. This user-programmable ID register is basically a user’s notepad provided in Flash or SRAM cells on each device.

In the course of system development and production, the proliferation of PLD architectures and patterns can be significant. To further complicate the record-keeping process, design changes often occur, especially in the early stages of product development. The task of maintaining which pattern goes into what device for which socket becomes exceedingly difficult. Once a manufacturing flow has been set, it becomes important to “label” each PLD with pertinent manufacturing information, which is beneficial in the event of a customer problem or return. A USERCODE register is incorporated into devices to store such design and manufacturing data as the manufacturer’s ID, programming date, programmer make, pattern code, checksum, ISC data file CRC, PCB location, revision number, or product flow. This assists you with the complex chore of record maintenance and product flow control. In practice, the user-programmable USERCODE register can be used for any of a number of ID functions.

Within 32 bits available for data storage, you may find it helpful to define specific fields to make better use of the available storage. A field may use only one bit (or all bits), and can store a wide variety of information. The possibilities for these fields are endless, and their definition is completely up to you.

Even with the device’s security feature enabled, the USERCODE register can still be read. With a pattern code stored in the USERCODE register, you can always identify which pattern has been used in a given device. As a second safety feature, when a device is erased and re-programmed, the USERCODE identification is automatically erased. This feature prevents any situation in which an old USERCODE might be associated with a new pattern.

It is your responsibility to update the USERCODE when reprogramming. The USERCODE information is not included in the fuse map checksum reading.

Loading the USERCODE instruction makes the USERCODE available to be shifted out in the Shift-DR state of the TAP controller. The USERCODE register can be read while the device is in normal functional operation, allowing the device to be scanned while operating.

Programming Hardware

All programming specifications, such as the programming cycle and data retention, are guaranteed when programming devices over the commercial temperature range (0 to 70 degrees C). It is critical that the programming and bulk erase pulse width specifications are met by the programming platform to ensure proper in-system programming. The details of device programming are invisible to you if you use Lattice programming hardware and software.

Computer Hardware

Programming is most commonly performed on a PC or Linux system using the parallel port cable or the USB port cable.

Parallel Port Cable

The cable uses the parallel port of a PC or Linux system for in-system programming of all Lattice devices. Programmer generates programming signals from the parallel port and passes them through the cable to the JTAG port of the devices. With this cable and a connector on the printed circuit board, no additional components are required to program a device. Refer to the cable data sheet for more detailed specifications and ordering information.

Hardware design considerations for new boards include whether the hardware designer will be using boundary scan test operations or low-voltage (3.3 V–1.8 V) devices. In a system using 3.3 V devices, the cable version 2.0 should be used. This cable operates with either a 3.3 V or 5 V power source. In a system using 1.8 V devices, cable version 3.0 must be used. This cable operates with a power of 1.8 V to 5.0 V.

USB Port Cable

The USB port cable uses the USB port of a PC or Linux system for in-system programming of all Lattice devices. Programmer generates programming signals from the USB port and passes them through the USB port cable to the JTAG, Slave SPI, or I^2C port of the device. Be sure to use JTAGI2C Interface Programming mode with the USB cable for the I2C port.

Programming Software

Programmer supports programming of all Lattice devices in a serial daisy chain programming configuration in a PC or Linux environment. The software is built around a graphical user interface. Any required data files are selected by browsing with a built-in file manager. The software supports serial a programming of all Lattice devices. Any non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed after their instruction register lengths are defined in the chain description. Any non-Lattice devices that are compliant with IEEE 1532 can be programmed using an IEEE 1532-compliant BSDL and ISC data file. Programmable devices from other vendors can be programmed through the vendor supplied SVF file.

Embedded Programming

Programmer embedded source code is available for programming devices in an embedded or customized environment. The programming source code is written in ANSI-standard C language, which can be easily incorporated into an embedded system or tester software to support programming of devices. This code supports such common operations as Erase, Program, Verify, and Secure. After completion of the logic design and creation of a bitstream file, Programmer can create the data files required for in-system programming on customer-specific hardware: PCs, testers, or embedded systems.

FPGA Configuration

Programmer provides efficient and economical alternatives to large and expensive PROMs that are normally used for configuring FPGA devices.

Because SRAM-based FPGA devices are volatile, they require reconfiguration on power cycles. This means that external configuration data must be held in a non-volatile device. On systems that require quick configurations or that do not have processor resources readily available, a dedicated serial PROM can be used. But such a PROM has to be large to accommodate large FPGA devices or multiple devices.

A much easier solution is to use a low-cost, industry-standard flash memory device combined with a CrossLink-NX device.

Serial Peripheral Interface Flash

Programmer, combined with a Lattice cable download, supports the programming of Serial Peripheral Interface (SPI) flash devices.

Several Lattice FPGAs can be configured directly from an external serial peripheral interface (SPI) flash memory devices. Because of their bitstream compression capability, these Lattice FPGAs allow the use of smaller-capacity SPI memory devices.

For an up-to-date list of Lattice devices that can be configured using SPI flash, as well as a list of supported SPI flash devices, refer to the topic “Serial Peripheral Interface (SPI) Flash Support” in the Lattice Radiant online help.

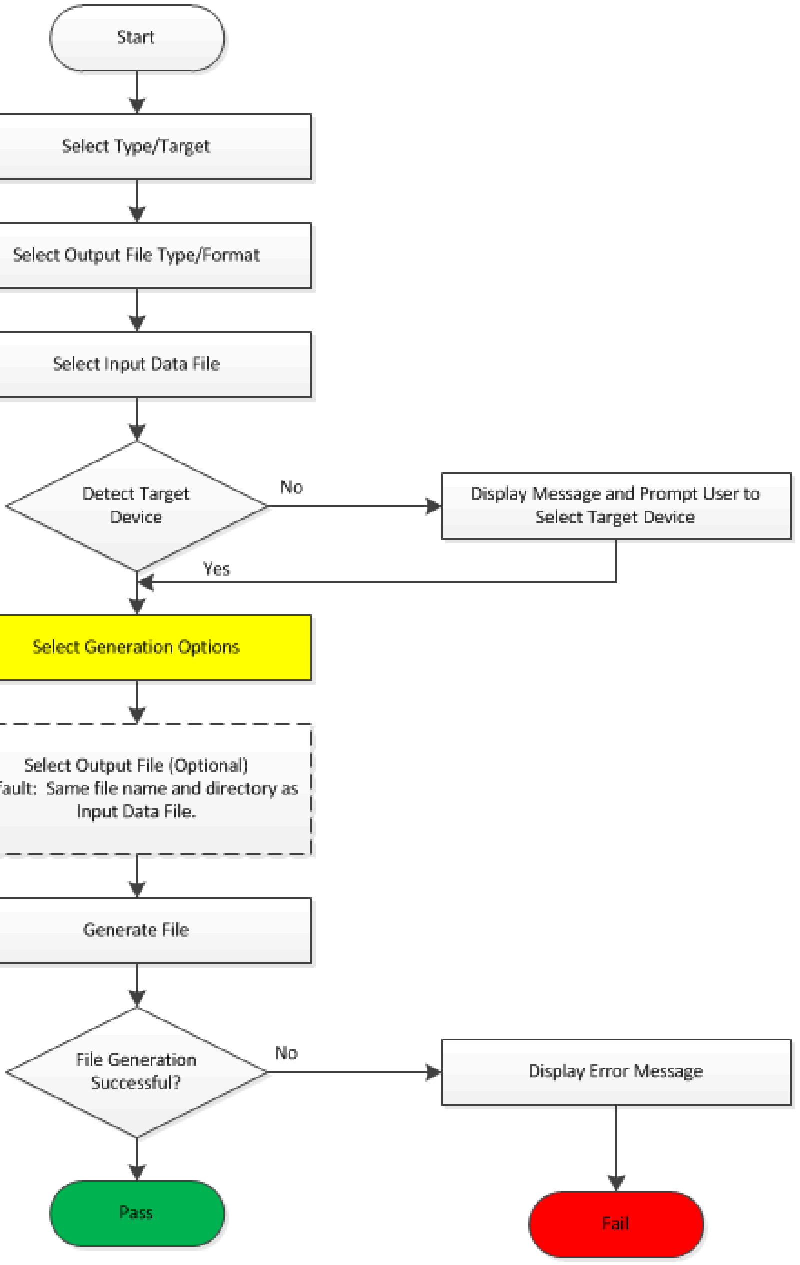
Deployment Tool Overview

The Deployment Tool graphical user interface (GUI) is separate from the Radiant and Programmer design environment, and is launched from Radiant Programmer by choosing **Tools > Deployment Tool**.

The Deployment Tool allows you to generate files for deployment for single devices, a chain of devices, and can also convert data files to other formats and use the data files it produces to generate other data file formats. A four-step wizard allows you to select deployment type, input file type, and output file type.

A basic block diagram of the Deployment Tool flow is shown in Figure 3 on page 20.

Figure 3: Deployment Tool Flow


Start
Select Type/Target
Select Output File Type/Format
Select Input Data File
Detect Target Device
No
Display Message and Prompt User to Select Target Device
Yes
Select Generation Options
Select Output File (Optional)
Default: Same file name and directory as Input Data File.
Generate File
File Generation Successful?
No
Display Error Message
Pass
Fail

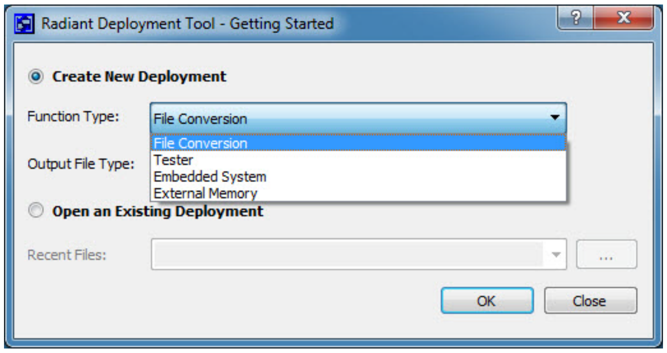
Deployment Tool Function Types

There are four function types available in Deployment Tool:

- File Conversion
- Tester
- Embedded System
- External Memory

The function types are accessed from the Function Type dropdown menu on the Deployment Tool Getting Started dialog box, as shown in Figure 4.

Figure 4: Deployment Tool Function Types



Output File Types

Each function type outputs different file types. This section describes all of the file types that are output by the five function types.

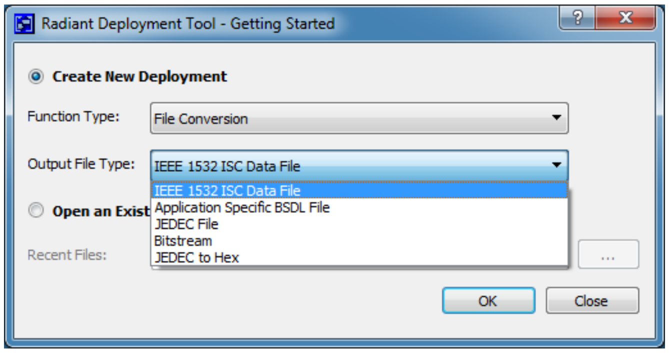
File Conversion Output File Types

The File Conversion function outputs four different file types, as shown in Figure 5 on page 22. The output types are defined as follows:

IEEE 1532 ISC Data File Converts JEDEC files to IEEE 1532 compliant ISC (In System Configuration) data files, which are used in conjunction with IEEE 1532 compliant BSDL files to program a device.

Application Specific BSDL File Converts a generic BSDL (Boundary Scan Description Language) file to an Application Specific BSDL file, using the signal names from the input file (JEDEC or ALT file). Also, for any I/Os that support VREFs or LVDS pairs and are configured as VREFs or LVDS pairs, the application-specific BSDL file changes to accurately reflect the behavior of the VREF or LVDS pair. When generating the Application Specific BSDL file, you have the option to convert bi-directional I/O's to inputs or outputs based

Figure 5: File Conversion Output File Types



on your design, or to keep all I/Os as bi-directional. The generic BSDL files are available on the Lattice website.

JEDEC File Converts the following file types JEDEC, Binary Bitstream, ASCII Bitstream, or IEEE 1532 ISC into a JEDEC file. The USERCODE, USERCODE format, and set the Program Security Fuse for the JEDEC file.

Bitstream Takes a Binary Bitstream, or ASCII Bitstream file and can convert it into the following output formats Binary Bitstream, ASCII Bitstream, Intel Hex, Motorola Hex, and Extended Tektronix Hex. Users can specify the Program Security Bit, Verify ID Code, Frequency, Compression, CRC Calculation, USERCODE format, and USERCODE.

JEDEC to Hex Converts JEDEC (*.jed) file type to either ASCII Raw Hex (*.hex) or Binary Raw Hex (*.bin) file type.

Note

The **JEDEC to Hex** feature supports JEDEC files generated by Lattice software. Using self-modified JEDEC files, corrupted JEDEC files, or JEDEC files generated using other software may result in incorrect data being generated, hanging, or crashing.

This feature does not support the following:

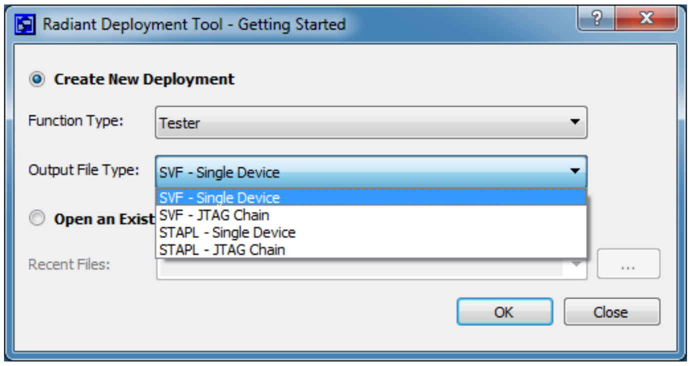
- Encrypted JEDEC files
- SED CRC
- TAG Memory
- USERCODE
- Feature Row

Refer to the Deployment Tool online help for information about specific device support.

Tester Output File Types

The Tester function outputs five different file types, as shown in Figure 6.

Figure 6: Tester Output File Types



The output types are defined as follows:

SVF - Single Device SVF Single Takes one of the following user data files types ASCII Bitstream, Binary Bitstream, or IEEE 1532 ISC and then select an operation to generate an SVF (Serial Vector Format) file. Depending on the data file selected then a certain set of operation for the device are available to be selected. The user is able to check several options which will modify the SVF file.

SVF - JTAG Chain Takes an XCF file generated by Programmer and generates an SVF file. There are several options available that modify the SVF file including write header and comments, and set maximum data size per row.

STAPL - Single Device Takes an ASCII Bitstream, Binary Bitstream, or IEEE 1532 ISC and then depending on the input file type gives a set of available operation that can be performed on the device. A STAPL (Standard Test And Programming Language) file is generated using the data file and operation.

STAPL - JTAG Chain Generates a STAPL file for testing using only an XCF file generated by Programmer.

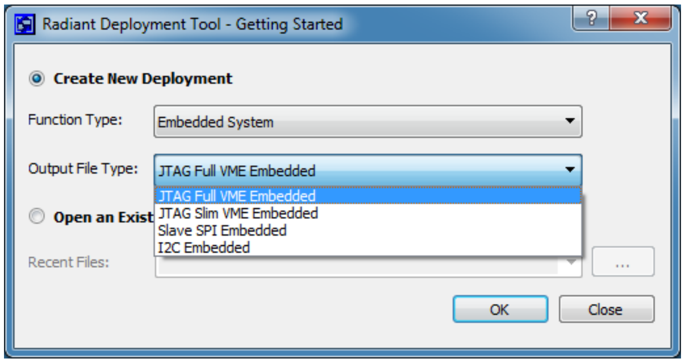
Refer to the Deployment Tool online help for information about specific device support.

Embedded System Output File Types

The Embedded System function outputs five different file types, as shown in Figure 7.

The output types are defined as follows:

Figure 7: Embedded System Output File Types



JTAG Full VME Embedded Takes an XCF as an input file, then the user can check options such as Compress VME, include Header along with several other options. This operation generates a VME file which is a compressed hexadecimal representation of an SVF files.

JTAG Slim VME Embedded VME is a compressed version of a VME file. To generate a Slim VME file an XCF file must be specified, then specify whether it is a Compressed VME file and whether or not to generate a HEX file. This operation outputs an algorithm VME file and a data VME file.

Slave SPI Embedded This file type allows field upgrades via the slave SPI port. This operation can be given an XCF, Binary Bitstream, and ASCII Bitstream as an input file. If an Bitstream file is given then the operation for the device must be specified along with whether or not to compress the embedded file and whether or not to generate a HEX file. If an XCF file is given there are no other operations or options the user needs to provide. This operation will output an algorithm file (.sea) and a data file (.sed).

I^{2}C Embedded I^{2}C embedded files enable field upgrades via the I^{2}C port. If an XCF file is specified then the user is given the option to compress the embedded files, generate a hex file, include comments, and if there should be a fixed pulse width. If a Bitstream file is specified then the previous options are available along with selecting the device operation and specifying the length of the I^{2}C Slave Address. Two files will be generated a data file (.ied) and an algorithm file (.iea).

Refer to the Deployment Tool online help for information about specific device support.

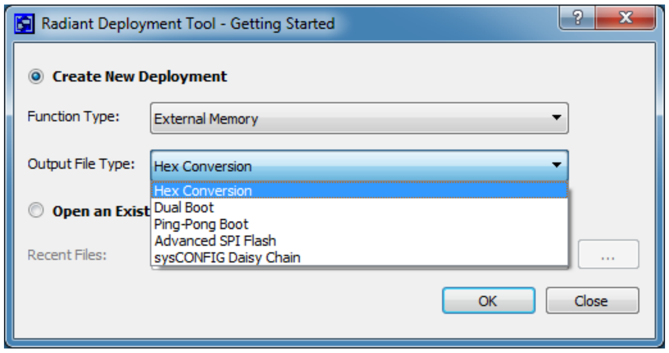
Also, refer to “Embedded Flow Overview” on page 27.

External Memory Output File Types

The External Memory function outputs four different file types, as shown in Figure 8 on page 25. The output types are defined as follows:

Hex Conversion converts a file Binary Bitstream, ASCII Bitstream, Binary, or Hex to various Hexadecimal file formats which are used to configure the

Figure 8: External Memory Output File Types



external SPI Flash memory of a device. The output file formats are Intel Hex, Motorola Hex, and Extended Tektronix Hex. The user is also able to set the Program Security bit, Verify ID Code, Frequency, compression, CRC Calculations and also the Starting Address.

Dual Boot Takes two Binary Bitstream or ASCII Bitstream files and then creates a single hex file to configure primary and golden sectors of an external SPI Flash. The output format can be Intel Hex, Motorola Hex, and Extended Tektronix Hex. The device will usually boot form the primary sector unless there is a problem then it will boot from the gold sector.

Advanced SPI Flash This operation is for generating hex files which handles more complicated operations such as Multiple Boot, and Quad I/O to configure external memory. Users can set the output hex format, how big the SPI Flash size is, whether or not to do a byte wide bit mirror, retain the bitstream header, and Whether or not to optimize the memory space. Another option is to set multiple user data file and where each of those data file's starting address should be in memory.

sysCONFIG Daisy Chain. This is used when multiple devices are in a daisy chain and configured from a single SPI flash or CPU. This operation will take two Binary or ASCII bitstreams and convert them into a single hex file.

Refer to the Deployment Tool online help for information about specific device support.

Embedded Flow Overview

Lattice Embedded VME enables in-field upgrades of Lattice programmable devices by suitable embedded processors, and consists of the following:

JTAG Full VME Embedded Enable field upgrades via the JTAG port.

JTAG Slim VME Embedded Featured s reduced foot print and is designed for microcontrollers with limited resources, such as 8051 processors.

Slave SPI Embedded Enable field upgrades via the slave SPI port.

I2C Embedded Enable field upgrades via the I^2C port.

There are three components to Embedded VME

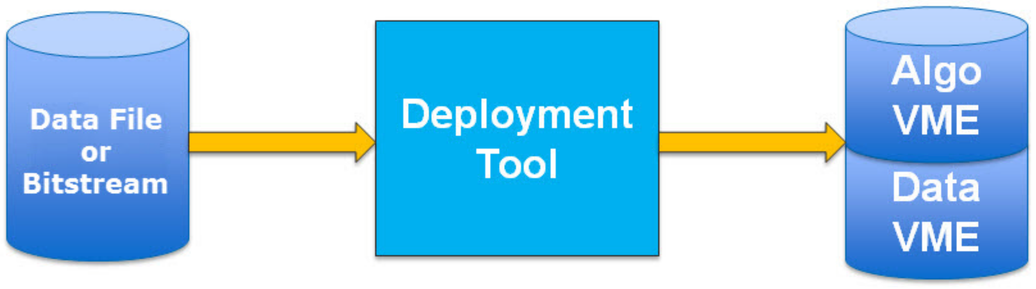
- ANSI C source code, which is shipped with Radiant Programmer. The user compiles this ANSI C Source code into their target system.
- Algorithm VME File, which contains the programming algorithm for the target FPGA. The Algorithm VME file is generated using the Deployment Tool.
- Data VME File, which contains the data that will be programmed into the FPGA. The Data VME file is generated using the Deployment Tool.

For all four embedded types, the Embedded VME support is comprised of C source files that users must port into their embedded systems for the purpose of programming Lattice devices. The porting process is also known as the customization and compiling process. The end product of the porting process will be the Embedded VME in compiled form, which will reside in the embedded systems.

Depending on the port interface, such as JTAG, SPI, or I^2C, the user can select one of the four embedded VME types.

Figure 9 shows an example of Full VME embedded file generation for the JTAG port.

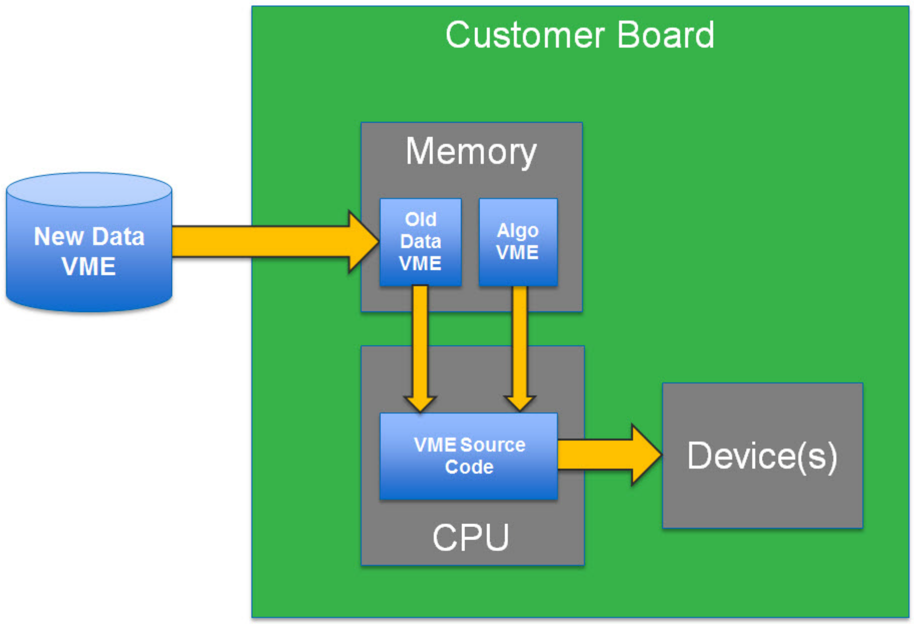
Figure 9: Full Embedded VME Flow



The programming data and programming instructions are compiled into a binary VME file format for the driver to load into the target devices. The VME file can be provided to the driver as a stand-alone file or linked together with the driver.

Figure 10 shows a high-level example of a file-based embedded VME used for field upgrades.

Figure 10: Example Embedded VME Programming Configuration



Porting of the JTAG VME into Embedded Systems

Porting JTAG Embedded VME is simple and the requirements are very simple to follow:

AC Requirements:

- TCK Fmax = 25 MHz.
- TCK Rise Time and Fall Time = 50ns maximum.
- Delay function resolution and accuracy = 1 millisecond minimum.

DC Requirements:

- I/O voltage level of the driver = I/O voltage level of the VCC JTAG port of the target devices. The VCC that power the JTAG port can be:
  - VCC core (All EE based devices)
  - VCCIO
  - VCCJ (All SRAM based and Flash based FPGA devices)
- Programming current = 1 Ampere maximum.

JTAG Programmability of Lattice Devices

Lattice's devices can be classified into three groups based on programmability:

- SRAM based only devices (volatile devices).
- EE based devices (non-volatile devices).
- Flash based devices (non-volatile devices).

Note

For information on configuring the Lattice iCE40 family of devices from an embedded processor, refer to TN1248, *iCE40 Programming and Configuration Guide*.

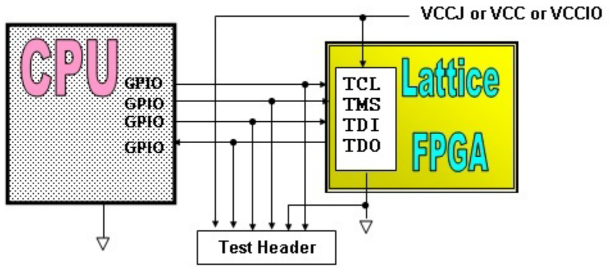
The SRAM based only devices are the easiest devices to support in terms of Embedded VME porting for they normally do not require accurate timing.

Embedded VME Porting Detail

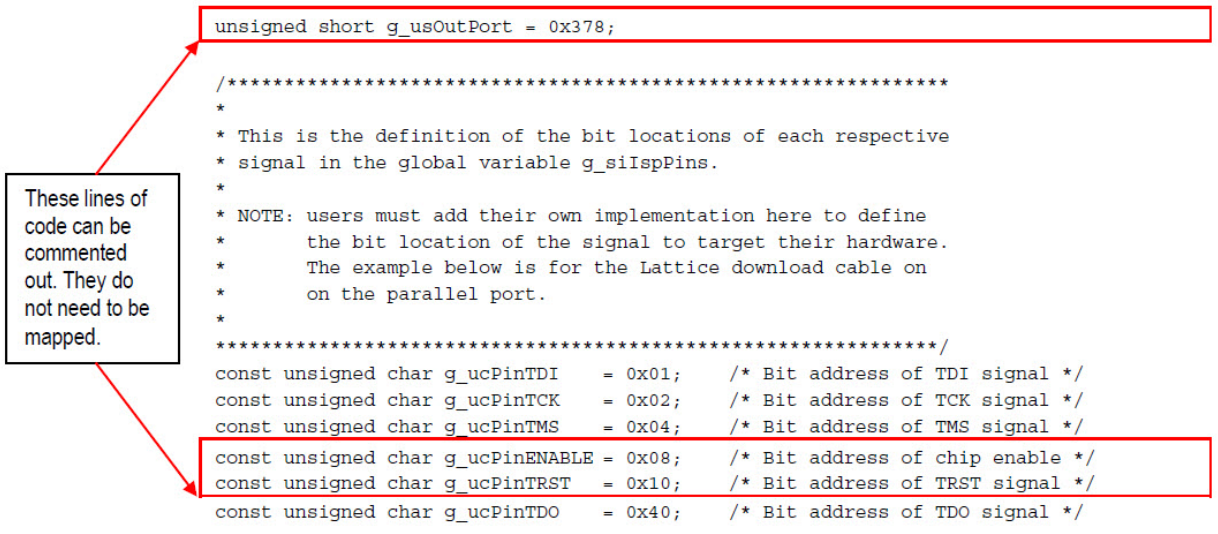
Step 1: Customize JTAG Embedded VME by modifying hardware.c

The pin mapping index table on the hardware.c must be revised to match with the customer’s board layout. On the PCB that is the target for porting the Embedded VME, it is important and a good practice to route the JTAG port to a test header for easy access using an oscilloscope or connecting to Programmer for debugging.

Figure 11: Map Four GPIO Pins from the CPU to the Four JTAG Pins



All VME files begin with IDCODE verification to ensure the JTAG port pins are mapped and connected properly.


These lines of code can be commented out. They do not need to be mapped.

```
unsigned short g_usOutPort = 0x378;

/*****************************************************************
*
* This is the definition of the bit locations of each respective
* signal in the global variable g_siIspPins.
*
* NOTE: users must add their own implementation here to define
*       the bit location of the signal to target their hardware.
*       The example below is for the Lattice download cable on
*       on the parallel port.
*
*****************************************************************/
const unsigned char g_ucPinTDI    = 0x01;    /* Bit address of TDI signal */
const unsigned char g_ucPinTCK    = 0x02;    /* Bit address of TCK signal */
const unsigned char g_ucPinTMS    = 0x04;    /* Bit address of TMS signal */
const unsigned char g_ucPinENABLE = 0x08;    /* Bit address of chip enable */
const unsigned char g_ucPinTRST   = 0x10;    /* Bit address of TRST signal */
const unsigned char g_ucPinTDO    = 0x40;    /* Bit address of TDO signal */
```

Modify the Delay Function

When porting Embedded VME to a native CPU environment, the speed of the CPU or the system clock that drives the CPU is usually known. The speed or the time it takes for the native CPU to execute one loop then can be calculated.

The for loop usually is compiled into the ASSEMBLY code as shown below:

```
LOOP: EDC RA;
        JNZ LOOP;
```

If each line of assembly code needs four (4) machine cycles to execute, the total number of machine cycles to execute the loop is 2 x 4 = 8.

Usually: system clock = machine clock (the internal CPU clock).

Note

Some CPUs have a clock multiplier to double the system clock for the machine clock.

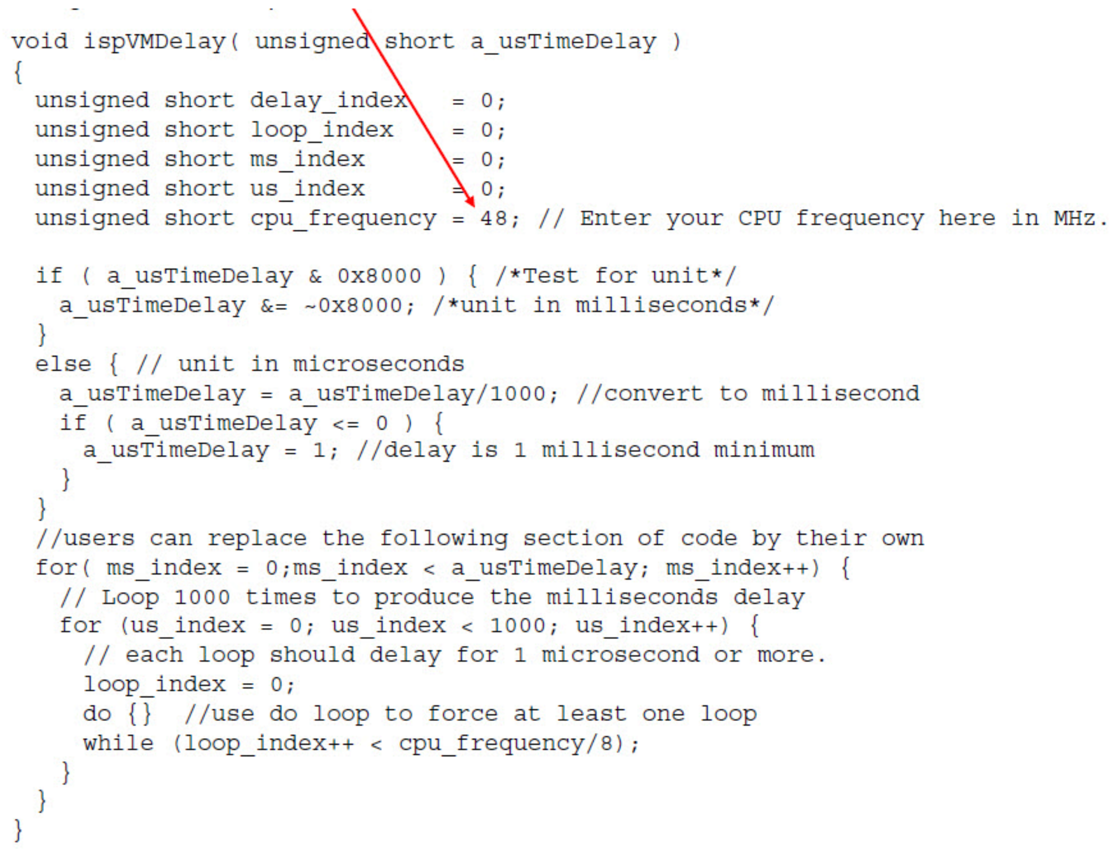
Let the machine clock frequency of the CPU be F (in MHz), then one machine cycle = 1/F.

The time it takes to execute one loop = (1/F) x 8.

It is obvious that the formula can be transposed into one microsecond = F/8.

Example: The CPU internal clock is set to 48 MHz, then one microsecond = 48/8 = 6.

The C code shown below can be used to create the millisecond accuracy. All that needs to be changed is the CPU speed.



```
void ispVMDelay( unsigned short a_usTimeDelay )
{
  unsigned short delay_index   = 0;
  unsigned short loop_index    = 0;
  unsigned short ms_index      = 0;
  unsigned short us_index      = 0;
  unsigned short cpu_frequency = 48; // Enter your CPU frequency here in MHz.

  if ( a_usTimeDelay & 0x8000 ) { /*Test for unit*/
    a_usTimeDelay &= ~0x8000; /*unit in milliseconds*/
  }
  else { // unit in microseconds
    a_usTimeDelay = a_usTimeDelay/1000; //convert to millisecond
    if ( a_usTimeDelay <= 0 ) {
      a_usTimeDelay = 1; //delay is 1 millisecond minimum
    }
  }
  //users can replace the following section of code by their own
  for( ms_index = 0;ms_index < a_usTimeDelay; ms_index++) {
    // Loop 1000 times to produce the milliseconds delay
    for (us_index = 0; us_index < 1000; us_index++) {
      // each loop should delay for 1 microsecond or more.
      loop_index = 0;
      do {}  //use do loop to force at least one loop
      while (loop_index++ < cpu_frequency/8);
    }
  }
}
```

Step 2: Calibration

It is important to confirm if the delay function is indeed providing the accuracy required. It is also important to confirm the TCK frequency. As an example, we will estimate the minimum system clock frequency of the native CPU that does not require the TCK to be slowed down. The TCK could be generated by the following code.

```
writePort (g_ucPinTCK, 0x00);
writePort (g_ucPinTCK, 0x01);
```

Let the number of system clocks to execute one line of code = 8 clocks.

The total number of clock for one pulse = 2 x 8 = 16.

The total amount of time for one pulse = 1/F x 16.

Lattice devices TCK frequency max = 25 MHz.

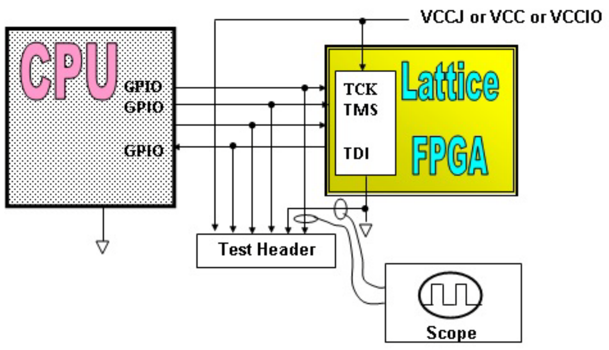
The equation becomes: 1/25 = 1/F x 16.

The maximum frequency of the CPU: F = 16 x 25 = 400 MHz.

If the system clock of the native CPU is faster than 400 MHz, the TCK pulses must be slowed down to meet the 25 MHz maximum specification.

The setup time and hold time of TDI, TMS, and TDO relative to TCK is not of concern for Embedded VME is constructed in the fashion that it is not possible to violate that requirement whenever the frequency of TCK is within the specification.

Figure 12: JTAG Embedded VME Delay Calibration

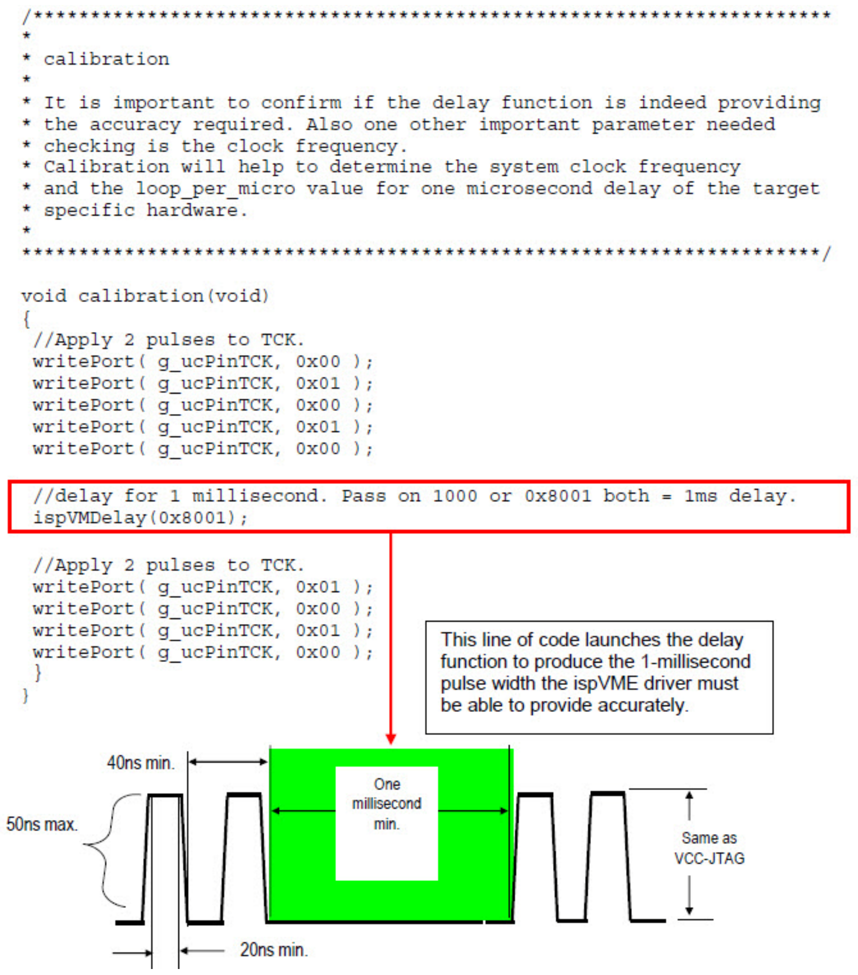


The calibrate function in Embedded VME can be launched by using the –c switch to cause the waveform as follow captured on the scope with the probe attached to the TCK wire.

If the pulse width is found to be smaller than 1 millisecond, then increase the cpu_frequency value until 1 millisecond delay is captured by the calibration function.

If the TCK frequency is found to be faster than 25 MHz, then change the sclock() function in hardware.c as shown below. The IdleTime normally is

Figure 13: JTAG Embedded VME Delay Calibration TCK Waveforms



```
/**********************************************************************
*
* calibration
*
* It is important to confirm if the delay function is indeed providing
* the accuracy required. Also one other important parameter needed
* checking is the clock frequency.
* Calibration will help to determine the system clock frequency
* and the loop_per_micro value for one microsecond delay of the target
* specific hardware.
*
**********************************************************************/

void calibration(void)
{
 //Apply 2 pulses to TCK.
 writePort( g_ucPinTCK, 0x00 );
 writePort( g_ucPinTCK, 0x01 );
 writePort( g_ucPinTCK, 0x00 );
 writePort( g_ucPinTCK, 0x01 );
 writePort( g_ucPinTCK, 0x00 );

 //delay for 1 millisecond. Pass on 1000 or 0x8001 both = 1ms delay.
 ispVMDelay(0x8001);

 //Apply 2 pulses to TCK.
 writePort( g_ucPinTCK, 0x01 );
 writePort( g_ucPinTCK, 0x00 );
 writePort( g_ucPinTCK, 0x01 );
 writePort( g_ucPinTCK, 0x00 );
 }
}
```

This line of code launches the delay function to produce the 1-millisecond pulse width the ispVME driver must be able to provide accurately.

initialized to 0. If it is initialized to 1, then the TCK frequency is effectively reduced by half. Use this technique to reduce the TCK frequency until meeting the specification.

```
/*******************************************************************************
* sclock
*
* Apply a pulse to TCK.
*
* This function is located here so that users can modify to slow down TCK if
* it is too fast (> 25MHZ). Users can change the IdleTime assignment from 0 to
* 1, 2... to effectively slowing down TCK by half, quarter...
*
*******************************************************************************/

void sclock()
{
  unsigned short IdleTime    = 0; //change to > 0 if need to slow down TCK
  unsigned short usIdleIndex = 0;

  IdleTime++;
  for ( usIdleIndex = 0; usIdleIndex < IdleTime; usIdleIndex++ ) {
    writePort( g_ucPinTCK, 0x01 );
  }

  for ( usIdleIndex = 0; usIdleIndex < IdleTime; usIdleIndex++ ) {
    writePort( g_ucPinTCK, 0x00 );
  }
}
```

Initialize to 1 if need to reduce TCK speed by half.

Step 3: Program Devices

Once the calibration is done, the Embedded VME (actually the JTAG port driver) is ready to program the devices. The device specific programming information is all self-contained in the VME file.

The VME file actually has six major sections:

1. Check the IDCODE,
2. Erase the device,
3. Program the device,
4. Verify the device,
5. Program the done fuse,
6. Wake-up the device.

IDCODE check failure is the most common failure when porting Embedded VME. It is a good practice to generate a Verify IDCODE only VME file first. Run the VME file. If it passes, then the JTAG port to GPIO mapping is confirmed. Once the port mapping is confirmed, then the programming VME file can be used.

Accurate timing is very critical to program devices reliably.

Using the calibration routine provided by Embedded VME will achieve the accurate timing.

JTAG Full VME Embedded

The JTAG Full VME Embedded VME software brings programming software to embedded applications. Using Lattice Semiconductor’s Radiant Programmer and Deployment Tools, you are provided with all necessary capabilities for programming devices in a single or multiple device chain. Developed to solve many programming issues facing today’s PLD users, JTAG Full VME Embedded provides advanced features including fast programming times, and small file sizes.

The JTAG Full VME Embedded software is a simplified version of the full Radiant Programmer. By making it serial vector format (SVF) file centric, JTAG Full VME Embedded is better targeted for embedded systems. Lattice JTAG devices are supported and users are able to program competitor devices through a simple SVF file translator. Lattice JTAG devices are those devices that can be programmed using the IEEE 1149.1 boundary scan TAP controller interface. Users are able to quickly and efficiently program chains of devices using this powerful utility, thus improving productivity and lowering costs.

An advantage of JTAG Full VME Embedded over vendor or architecture-specific methods is that once it is developed, it supports all present and future devices. As long as the programming flow can be described as an SVF file, the main engine does not have to change. For embedded environments, it is important to have deterministic memory requirements. By pre-processing the SVF file, it is possible to know the exact resources required to implement the programming algorithm and to store the programming data. The nature of the SVF file also allows the resources available to determine how the file is processed. Large shift instructions can be broken into multiple instructions if the embedded system does not have enough RAM available to store the entire row in one pass. Since the SVF file is serial in nature, it can be segmented to fit available RAM, PROM or FLASH memory.

The JTAG Full VME Embedded source code is designed to be hardware and platform independent. A VME data file, or VME file, runs on all JTAG Full VME Embedded applications.

See Also ▶JTAG Full VME Embedded Basic Operation

VME File Format

A VME file is simply an SVF file that has been compressed. SVF file includes algorithm and data file in ASCII format, and VME file is the SVF file in the optimized binary format. Compared with SVF file, VME files require minimized memory space to store the bitstream file and has optimized code size. SVF keywords such as SIR and SDR are replaced with the byte codes 0x11 and 0x12, respectively. This reduces the VME file by writing only one byte of data, the byte codes, instead of writing the entire SVF keyword, which would use more characters.

JTAG Full VME Embedded file supports compression to reduce the VME file size by compressing the data and address streams. A looping compression is also employed to reduce the file size even further by taking advantage of the repeating SVF constructs. The following describes each compression scheme.

Compressed VME Files The compression scheme is applied to the address and data stream following SIR and SDR, respectively. These streams will try to be compressed by 0x00, 0xFF, or by 4-bit count.

For example, consider the following line in a SVF file:

```
SDR 102 TDI (20000000000000000000000000);
```

The address stream is ‘20000000000000000000000000’. The repeating zeros in the stream can be easily compressed by 0x00. Compression with 0xFF works in the same manner, except that instead of the data stream containing zeros, it would contain ‘F’s.

Compression by 4-bit count works by looking for repeating patterns within the data stream that are not zeros or ‘F’s.

For example, consider the following line in a SVF file:

```
SDR 80 TDI (7F97F97F97F97F97F97F9);
```

The repeating 4-bit count in this example would be ‘7F9,’ because it repeats throughout the data stream. The 4-bit would be written only once in the VME file, and would be followed by the number of repetitions found within the data stream.

The compression scheme reduces the file size by not extrapolating repeating information within the address and data streams. That task is left for the VME processor.

Looping VME Files In an SVF file, repeating constructs can be observed. The looping scheme takes advantage of these constructs by creating a template with the repeating information, and the differentiating date is replaced by a placeholder. The differentiating data will be written after the construct.

For example, the following data is found in a SVF file:

```
SIR 5 TDI (01);

SDR 102 TDI (20000000000000000000000000);

SIR 5 TDI (02);

SDR 80 TDI (7BFFF7BFFFF7BFFF7BFF);

SIR 5 TDI (07);

RUNTEST IDLE 3 TCK 1.20E-002 SEC;

SIR 5 TDI (01);
```

```
SDR 102 TDI (10000000000000000000000000);

SIR 5 TDI (02);

SDR 80 TDI (FFFF7FFFFFFFFFFFFFFF);

SIR 5 TDI (07);

RUNTEST IDLE 3 TCK 1.20E-002 SEC;

SIR 5 TDI (01);

SDR 102 TDI (08000000000000000000000000);

SIR 5 TDI (02);

SDR 80 TDI (FFFFFFFFFFFFFFFFFFFF);

SIR 5 TDI (07);

RUNTEST IDLE 3 TCK 1.20E-002 SEC;

...

...
```

The looping template is built based on the repeating SIR lines. Notice how the TDI values for the SIR commands are a repeating sequence of 01, 02, and 07. In this case the resulting template would be:

```
SIR 5 TDI (01);

SDR 102 TDI VAR;

SIR 5 TDI (02);

SDR 80 TDI VAR;

SIR 5 TDI (07);

RUNTEST IDLE 3 TCK 1.20E-002 SEC;
```

VAR is written in place to hold the data that does not repeat. The non-repeating data will get written into the VME file following each template. The example above would look like this in the VME file:

```
LOOP 3

SIR 5 TDI (01);

SDR 102 TDI VAR;

SIR 5 TDI (02);
```

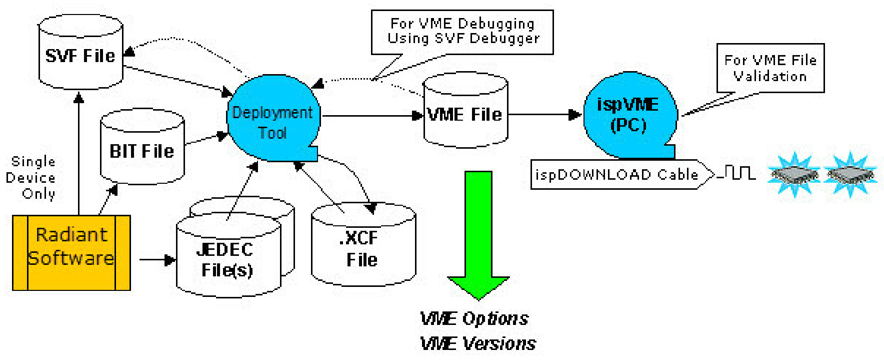
```
SDR 80 TDI VAR;

SIR 5 TDI (07);

RUNTEST IDLE 3 TCK 1.20E-002 SEC;

ENDLOOP

(20000000000000000000000000)

(7BFFF7BFFFF7BFFF7BFF)

(10000000000000000000000000)

(FFFF7FFFFFFFFFFFFFFF)

(08000000000000000000000000)

(FFFFFFFFFFFFFFFFFFFF)
```

The ‘LOOP 3’ tells the VME processor to loop the template three times. Each time it encounters a ‘VAR’, it will grab the first available line of data following the ‘ENDLOOP’ and replace ‘VAR’ with it. This technique reduces the file significantly by keeping the similar constructs to a minimal, and only writing the differences.

JTAG Full VME Embedded Flow

The JTAG Full VME Embedded System allows you to program a device using the microprocessor in an embedded system. When you install the VM software, a separate VMEmbedded folder containing the VME source code and executables is installed on your hard drive. Compiling the VME source code gives you an executable file that you can store in your system’s memory for programming using the JTAG port.

The following figure illustrates the JTAG Full VME Embedded flow.

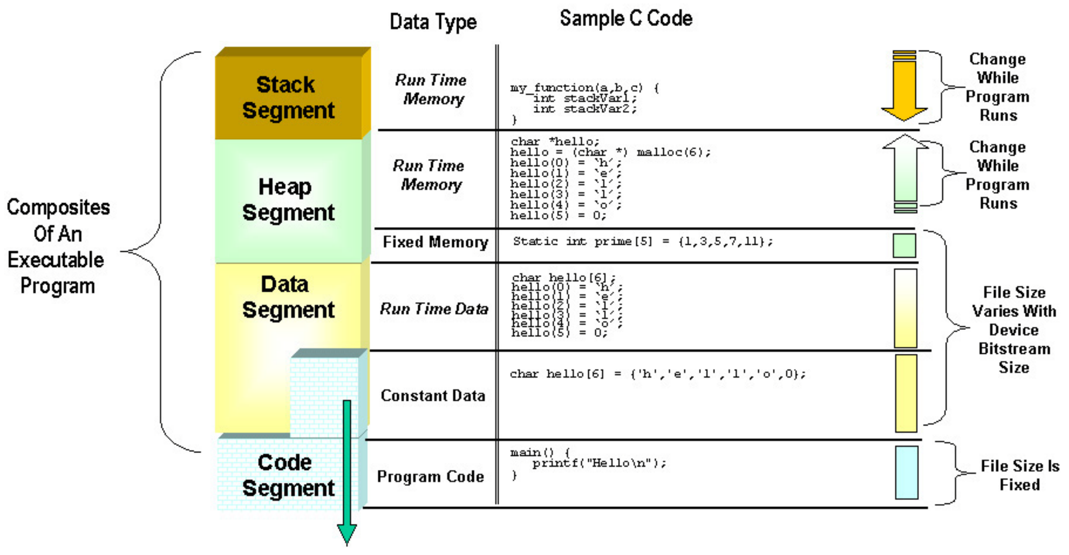


See Also ▶Generating VME Files

▶ Testing VME Files

JTAG Full VME Embedded System Memory

The following figure illustrates JTAG Full VME Embedded system memory.



```
my_function(a,b,c) {
   int stackVar1;
   int stackVar2;
}
```

```
char *hello;
hello = (char *) malloc(6);
hello(0) = `h´;
hello(1) = `e´;
hello(2) = `l´;
hello(3) = `l´;
hello(4) = `o´;
hello(5) = 0;
```

```
Static int prime[5] = {1,3,5,7,11};
```

```
char hello[6];
hello(0) = `h´;
hello(1) = `e´;
hello(2) = `l´;
hello(3) = `l´;
hello(4) = `o´;
hello(5) = 0;
```

```
char hello[6] = {'h','e','l','l','o',0};
```

```
main() {
   printf("Hello\n");
}
```

The Compiler can be directed to link the *Constant Data* to the *Code Segment* to reduce RAM usage

The *Code Segment* is typically placed in a non-volatile memory (PROM or FLASH)

The *Data Segment* contains data structures that have been preallocated storage.

See Also ▶JTAG Full VME Embedded Basic Operation

JTAG Full VME Embedded Basic Operation

There are three modes of JTAG Full VME Embedded operation.

File Mode Under the file mode, data is stored in a file system such as a hard drive or a DOS flash. The data file is accessed using C library calls, such as fopen, fread, and fclose. The file read operations collect data into the system memory. The system memory of the Embedded system must be able to store the entire bitstream from the file in a contiguous block of memory. The memory block can be allocated in one of the three locations.

- Data Segment – You can pre-determine how many bytes of data the bitstream will require and then create an uninitialized array variable to hold the data. This permanently allocates a portion of the Data Segment. For example:

  ```
  char programmingData[0x10000]; // allocate 64K
  ```

- Stack Segment – You can pre-determine how many bytes of data the bitstream will require and then create an uninitialized array variable to hold the data. Depending upon the function call sequence, this may or may not permanently allocate a portion of the system memory. See the example code below.

```
int MyFunction ( ) {
char bitstreamArray[0x10000];
}
```

- Heap Segment – You can determine at runtime how many bytes of data the bitstream will require and then dynamically allocate an uninitialized array variable to hold the data. You are responsible for freeing the memory when it is not being used any longer. Below is an example.

```
char *bitstreamData;
bitstreamData = (char *)malloc(numberOfBitstreamBytes);
```

Static Linking Mode Under the static linking mode, the bitstream data is converted from the file on the hard drive into a C source code file. The C source code defines a byte array. The byte array is exactly the size of the bitstream. The byte array can be linked into either the Code Segment or the Data Segment. The memory allocated for the bitstream is permanently consumed.

PROM Mode Under the PROM-based mode, the bitstream file is converted from the file on the hard drive into an Intel HEX file. The HEX file is loaded into a non-volatile memory using a PROM programming tool. The HEX file data is placed in the non-volatile memory at a known address (that is, a fixed address). The user C code initializes a pointer. The pointer is given the starting address of the HEX byte stream. The memory used by the bitstream is permanently allocated in the non-volatile memory.

See Also ▶JTAG Full VME Embedded System Memory

VME Source Code

The JTAG Full VME Embedded source code is written in standard ANSI C and is simplified with embedded applications in mind. Most embedded applications have greater limits on program and data sizes than PC or workstation applications. The areas most likely to differ between platforms are the timing delay function and hardware port manipulation.

The current version of the JTAG Full VME Embedded software is available through the Programmer installation. The installation creates a sub-directory called **VMEmbedded**, where the pre-compiled executables, source code, and readme.txt can be found.

There are four sets of embedded-related source code that are shipped with Programmer.

- VME – The file-based VME is the programming engine that accepts VME files as command line arguments to process the devices. By default, the executable compiled from this source code targets Windows operating systems. You can make small modifications to make the compiled executable accommodate other platforms.

- VME_eprom – The PROM-based VME is the programming engine that requires compiling a HEX file, which is a C-programming file, with the source code to create an executable engine that can be embedded onto the embedded system.
- svf2vme – The svf2vme is a command line utility that can convert SVF files into VME files.
- vme2hex – The vme2hex is a command line utility that can convert VME files into HEX files.

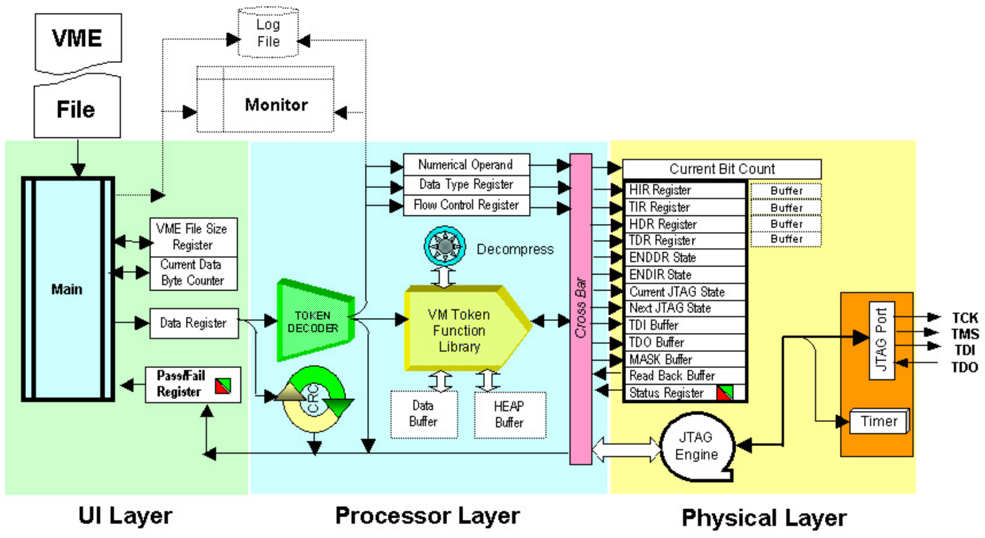
Among all the source codes, only the hardware.c file requires user changes. You should customize the hardware.c file according to your target platform.

JTAG Full VME Embedded Programming Engine

The programming engine of the JTAG Full VME Embedded software is driven by the byte codes of the VME format file. It manipulates the I/O ports and sends commands to the customer firmware. The commands sent from the programming engine requires the I/O system to be connected to the device's JTAG port. The VME byte codes instruct the engine as to what sequence of functions to follow in order to shift in instructions, move the TAP controller state machine, shift data in and out of the device, and observe delay. The engine has the following three layers.

- User interface layer (ispvm_ui.c) – Directs inputs and outputs.
- Processor layer (ivm_core.c) – Decodes commands, checks CRC prior to processing, and does optional decompression.
- Physical layer (hardware.c) – Shifts data to target device. This is the only file that you need to edit. See Customizing for the Target Platform for details.

The following figure illustrates JTAG Full VME Embedded JTAG port programming engine.



Legend	Size (bits)	Type
◺	8	signed char
◇	8	unsigned char
◺◺	16/32 (Note 1)	integer
◇◇	16	unsigned short integer

Legend	Description
┈	Optional features enabled by the VME, CPU or SCM file
🟥	Continue or Exit decision flag (Go / No Go)
☐	Memory allocated at run time using malloc() as required by the VME, CPU or SCM file.
▨	Gray out means not used.

RAM Size Requirement for VME

To calculate the worst-case size of memory needed to program a device, in terms of bytes, locate the size of the largest register in the device. This is usually the data shift register. Divide that number by eight, and then multiply the quotient by two: one for TDI and one for TDO. If the device has MASK, multiply the quotient by three instead of two.

This method only calculates the RAM requirements for the data of the device. It does not account for transient variables that are used to execute the programming algorithm. Transient variables are more difficult to calculate because they appear in and out of scope often. Also, a variable size may depend on the microprocessor’s register size. For example, an integer variable on a 32-bit system is four bytes while the same variable on a 16-bit system is only two bytes.

To approximate the RAM requirement for the run-time variables, add twenty percent to the required RAM.

To verify that the calculation is correct, convert the SVF file to VME, and use the VME2HEX utility to convert from VME to HEX. This utility generates the vme_file.h file, which gives the definitive memory size requirement.

The variables that are of concern to memory are:

- MaskBuf
- TdiBuf
- TdoBuf
- HdrBuf
- TdrBuf
- HirBuf
- TirBuf
- HeapBuf
- CRCBuf
- CMASKBuf

As expected, MaskBuf, TdiBuf, and TDOBuf each requires 26 bytes. If the device were in a chain, HdrBuf (Header Data Register), TdrBuf (Trailer Data Register), HirBuf (Header Instruction Register), and TriBuf (Trailer Instruction Register) would need extra bytes.

If the VME file had been generated with the looping option, HeapBuf would require extra bytes as well. Looping requires slightly more RAM but significantly less ROM. When the VME file has not been looped, it does not require any additional RAM, but ROM size can significantly increase. This trade-off is file-dependent.If the original SVF were 1532-compliant, CRCBuf and CMASKBuf would require extra bytes as well.

ROM Size Requirement for JTAG Full VME Embedded

To calculate the worst-case ROM size for a given device, multiply the number of frames by the frame size. Divide that number by eight to obtain the required ROM size, in terms of bytes.

This method assumes that the SVF file will be generated with the turbo option. If the SVF file were generated with the sequential option, the worst-case ROM size would be doubled.

This method only calculates the ROM requirements for the data. It does not account for opcodes that are used to translate the algorithm of the device. To approximate the ROM requirement for the algorithm opcodes, add twenty percent to the required ROM.

The actual ROM requirement might be significantly less than the theoretical worst-case requirement because SVF2VME utilizes two compression techniques, compression and looping, to decrease the VME file. The file size difference is file-dependent.

JTAG Full VME Embedded Required User Changes

To make the JTAG Full VME Embedded or JTAG Slim VME Embedded software work on your target system, you need to modify the following C functions in the hardware.c source code.

Timer The engine requires the ability to delay for fixed time periods. The minimum granularity of the delay is 1 microsecond. You can determine the type of delay. This can be a simple software timing loop, a hardware timer, or an operating system call, for example, sleep().

Port Initialization The firmware needs to place the port I/O into a known state. The software assumes this has occurred.

Get Data Byte The engine calls the GetByte() function to collect one byte from the JTAG Full VME Embedded or CPU bytestream.

Modify Port Register The engine, as it parses the bitstream data, changes an in-memory copy of the data to be written onto the I/O pins. Calls to this function do not modify the I/O pins. The engine uses *virtual types* (for example, DONE_PIN) which this function turns into physical I/O pin locations (for example, 0x400).

Output Data Byte The engine calls this function to write the in-memory copy onto the I/O pins.

Input Status This function is used by the engine to read back programming status information. The function translates physical pin locations (for example, 0x400) into *virtual types* used by the engine (for example, DONE_PIN).

Output Configuration Pins Some systems may wish to use the FPGA CFG pins, and have the Embedded system control them. There is a separate function call to manipulate the CFG pins.

Bitstream Initialization You must determine how you plan to get the bitstream into your memory system, pre-compiled, HEX file based, or dynamically installed. Whichever method you use the data structures which pin to the bitstream need to be initialized prior to the first GetByte function call.

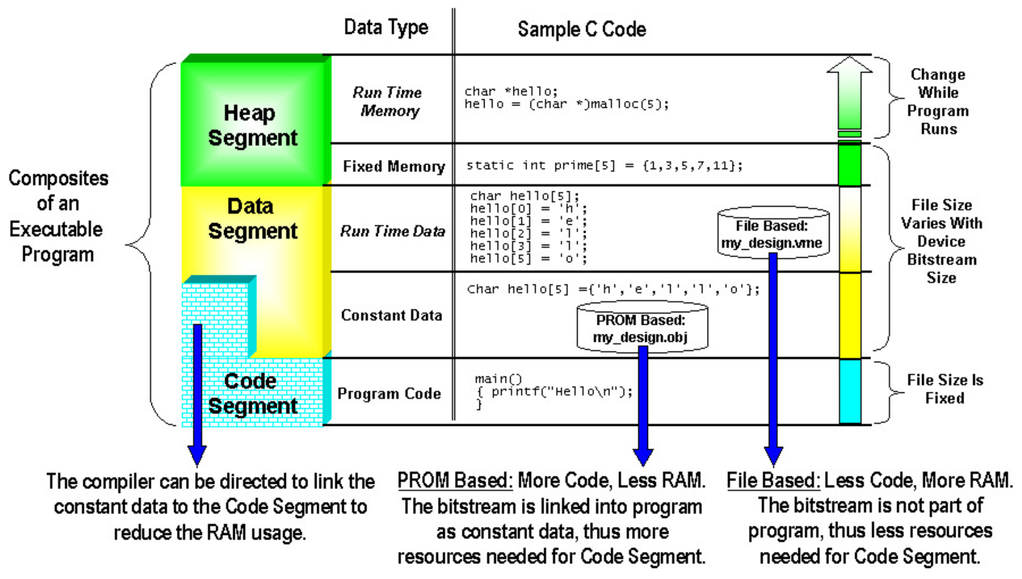
See Also ▶Customizing for the Target Platform

▶ VME Source Code

▶ JTAG Slim VME Embedded Source Code

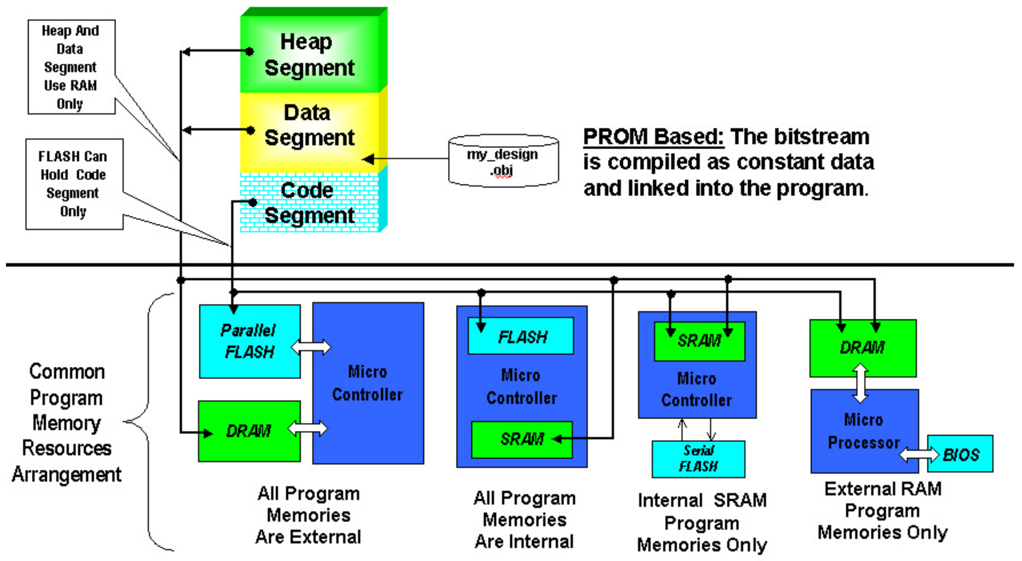
Program Memory Requirement

The following figure illustrates the JTAG Full VME Embedded program memory requirement.


Data Type
Sample C Code
Composites of an Executable Program
Heap Segment
Data Segment
Code Segment
Run Time Memory
char *hello;
hello = (char *)malloc(5);
Change While Program Runs
Fixed Memory
static int prime[5] = {1,3,5,7,11};
Run Time Data
char hello[5];
hello[0] = 'h';
hello[1] = 'e';
hello[2] = 'l';
hello[3] = 'l';
hello[5] = 'o';
File Based: my_design.vme
File Size Varies With Device Bitstream Size
Constant Data
char hello[5] ={'h','e','l','l','o'};
PROM Based: my_design.obj
Program Code
main()
{ printf("Hello\n");
}
File Size Is Fixed
The compiler can be directed to link the constant data to the Code Segment to reduce the RAM usage.
PROM Based: More Code, Less RAM. The bitstream is linked into program as constant data, thus more resources needed for Code Segment.
File Based: Less Code, More RAM. The bitstream is not part of program, thus less resources needed for Code Segment.

Program Memory Allocation

The following figure illustrates the JTAG Full VME Embedded program memory allocation.

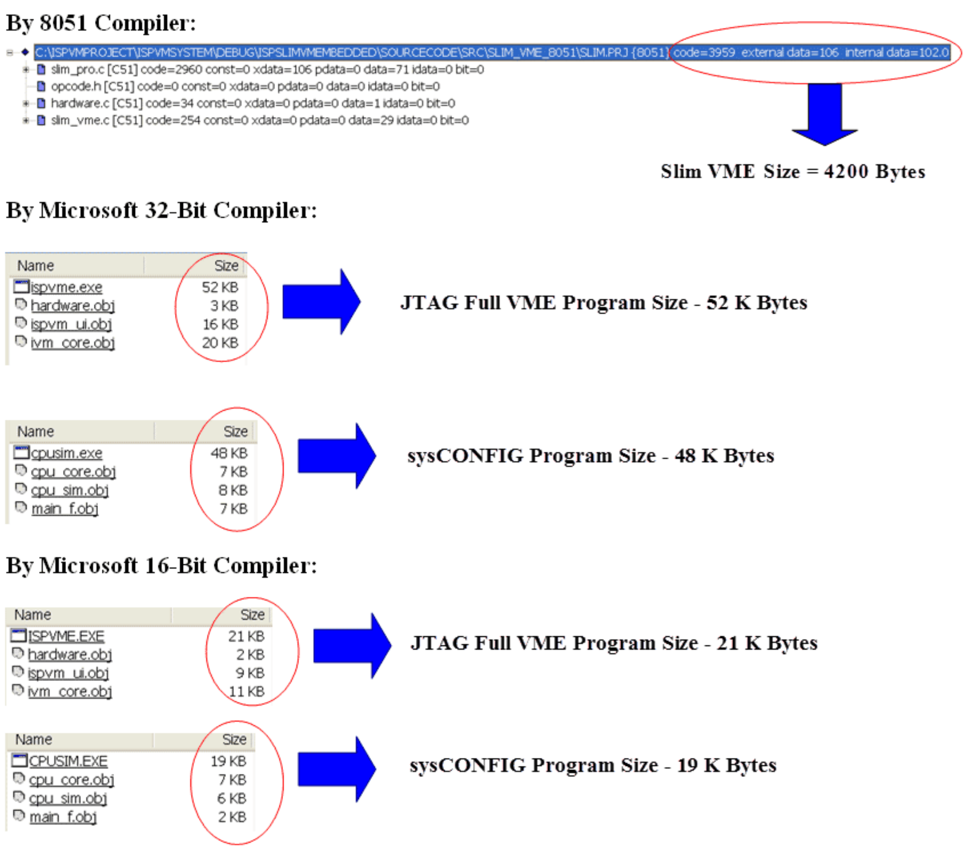

Heap And Data Segment Use RAM Only
FLASH Can Hold Code Segment Only
Heap Segment
Data Segment
Code Segment
my_design .obj
PROM Based: The bitstream is compiled as constant data and linked into the program.
Common Program Memory Resources Arrangement
Parallel FLASH
DRAM
Micro Controller
All Program Memories Are External
FLASH
Micro Controller
SRAM
All Program Memories Are Internal
SRAM
Micro Controller
Serial FLASH
Internal SRAM Program Memories Only
DRAM
Micro Processor
BIOS
External RAM Program Memories Only

Sample Program Size

This page provides sample program size for JTAG Full VME Embedded, JTAG Slim VME Embedded, and sysCONFIG Embedded.

Embedded Tool / Bitstream Location	JTAG Port			non-JTAG Port		Total
	JTAG Full VME Embedded		JTAG Slim VME Embedded	sysCONFIG Embedded		
	32-Bit	16-Bit	8-Bit	32-Bit	16-Bit	
File Based (Bitstream File External)	52KB	21KB	4.2KB	48KB	19KB	As Shown
PROM Based (Bitstream File Integrated)	52KB	21KB	4.2KB	48KB	19KB	As Shown + VME File Size

.



By 8051 Compiler:

```
C:\ISPVMPROJECT\ISPVMSYSTEM\DEBUG\ISPSLIMVMEMBEDDED\SOURCECODE\SRC\SLIM_VME_8051\SLIM.PRJ {8051} code=3959  external data=106  internal data=102.0
  slim_pro.c [C51] code=2960 const=0 xdata=106 pdata=0 data=71 idata=0 bit=0
  opcode.h [C51] code=0 const=0 xdata=0 pdata=0 data=0 idata=0 bit=0
  hardware.c [C51] code=34 const=0 xdata=0 pdata=0 data=1 idata=0 bit=0
  slim_vme.c [C51] code=254 const=0 xdata=0 pdata=0 data=29 idata=0 bit=0
```

Slim VME Size = 4200 Bytes

By Microsoft 32-Bit Compiler:

Name	Size
ispvme.exe	52 KB
hardware.obj	3 KB
ispvm_ui.obj	16 KB
ivm_core.obj	20 KB

JTAG Full VME Program Size - 52 K Bytes

Name	Size
cpusim.exe	48 KB
cpu_core.obj	7 KB
cpu_sim.obj	8 KB
main_f.obj	7 KB

sysCONFIG Program Size - 48 K Bytes

By Microsoft 16-Bit Compiler:

Name	Size
ISPVME.EXE	21 KB
hardware.obj	2 KB
ispvm_ui.obj	9 KB
ivm_core.obj	11 KB

JTAG Full VME Program Size - 21 K Bytes

Name	Size
CPUSIM.EXE	19 KB
cpu_core.obj	7 KB
cpu_sim.obj	6 KB
main_f.obj	2 KB

sysCONFIG Program Size - 19 K Bytes

Using JTAG Full VME Embedded

The procedure of generating and processing the VME can be done by using the Programmer graphical user interface.

Table 1: JTAG Full VME Program Descriptions

Program	Description
JTAG Full VME Embedded (file-based)	The file-based JTAG Full VME Embedded is the programming engine that accepts VME files as command line arguments to process the device(s).
JTAG Full VME Embedded (EPROM-based)	The EPROM-based JTAG Full VME Embedded is the programming engine that requires compiling a HEX file, which is a C-programming file, with the source code to create an executable engine that can be embedded onto the embedded system.
svf2vme	The svf2vme is a command line utility that can convert SVF files into VME files.
vme2hex	The vme2hex is a command line utility that can convert VME files into HEX files.

Generating VME Files

A VME file is a variation of an SVF file that has been compressed into a binary file. It allows you to program a device from the microprocessor on your printed circuit board. The VME files can be created in Deployment Tool by selecting Lattice Programmer-generated an XCF file. An XCF file is a configuration file used by Radiant Programmer and for programming devices in a JTAG daisy chain. The XCF file contains information about each device, the data files targeted, and the operations to be performed.

In Deployment Tool, the JTAG Full VME Embedded software will then take the device chain information and generate the VME file. If a non-Lattice device is in the chain, you must add a JTAG-SVF device and supply the SVF file. For chains with JTAG-SVF devices, JTAG Full VME Embedded generates two VME files. You can use one or both files to program the device.

To generate a VME file:

1. In Programmer, create a project, and add the target devices into the chain with the appropriate operations and data files. If a non-Lattice device is in the chain, set the device as a JTAG-SVF device and provide the appropriate SVF file, SVF vendor, and TCK frequency. Refer to Programmer online help for more information on how to use Programmer.
2. Save the Programmer project (.xcf).
3. In Deployment Tool, choose **Create New Deployment.**
4. For Function Type, choose **Embedded System**.

5. For Output Type, choose **JTAG Full VME Embedded**, then click **OK**.
6. In the Step 1 of 4 dialog box, select the XCF file, and click **Next.**
7. In the Step 2 of 4 dialog box, elect the desired file options. For detailed option descriptions, including the option that allows you to generate a HEX (.c) file, see the Deployment Tool online help.
8. Click **Next**.
9. In the Step 3 of 4 dialog box, in the Output File box, specify the location and file name of the VME file.
10. Click **Next**.
11. In the Step 4 of 4 dialog box, click **Generate**.

Deployment Tool generates the VME file depending upon the options you have chosen, and returns a message indicating that the process succeeded or failed.

Testing VME Files

Use the Download Debugger to process the VME file using any of the Lattice programming cables. Refer to Download Debugger online help for details. This processor can run through using the Lattice download cable.

VME files can also be processed using the command line. See Running the Deployment Tool from the Command Line online help for details.

Converting an SVF File to VME File

VME files can also be generated the traditional way by using the svf2vme source code. The utility will expect an SVF file as argument.

Choosing the File-Based or EPROM-Based Version

To generate a PROM-based VME, select the “Generate HEX (.c) File” option in the Deployment Tool Step 2 of 4 dialog box.The programming engines of the file-based and PROM-based processors are identical in the way they handle the VME commands. Their difference lies in the way they interface with VME data. For a convenient demo, the file-based version assigns a file pointer to the binary VME file directly. The pointer is assigned based on a command line argument. With some minor modification, this version is useful for embedded high-level 32-bit microprocessors that can dynamically allocate RAM and have large amounts of data and code memory. For more modest embedded systems or smaller processors, the PROM-based version is useful because the memory resources are completely defined when compiling the executable.The VME file is converted to one or more C files and a header file that are compiled with the core routines.

Customizing for the Target Platform

The main routines that will require customization are in the hardware.c file. They include the routines for reading from and writing to the JTAG pins and a delay routine. These routines are well commented in hardware.c and are at the top of the file. In readPort(), a byte of data is read from the input port. In writePort(), a byte of data is written to the output port. In ispVMDelay(), the system delays for the specified number of milliseconds or microseconds. The port mapping is set at the top of the hardware.c file.

The source code files are written in ANSI C. The JTAG Full VME Embedded source codes are located in the <*install_path*>\embedded_source directory.

See Also ▶JTAG Full VME Embedded Required User Changes

Advanced Issues

Since SVF files are serial in nature, many vendors have options on the type of operations to be performed when generating the SVF files. If an SVF file is too large for the targeted embedded application, consider removing optional operations or breaking up the operations by creating multiple SVF files. This approach is much better than arbitrarily dividing the VME file.

EPROM-based JTAG Full VME Embedded User Flow

This appendix details the steps the user must take to use the EPROM-based JTAG Full VME Embedded.

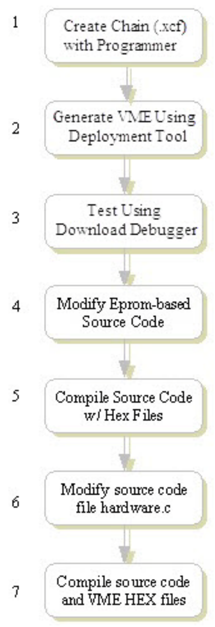
Step 1. Create Chain With Programmer

Using Programmer, add the target devices into the chain with the appropriate operations and data files. If a non-Lattice device is in the chain, set the device as a JTAG-SVF device and provide the appropriate SVF file, SVF vendor, and TCK frequency. For more information on supporting non-Lattice devices, see Programmer’s on-line help documentation.

Step 2. Generate VME File

Use the Deployment Tool to generate the VME file. Refer to the Deployment Tool online help for more information on Deployment Tool.

Figure 14: EPROM-based JTAG Full VME Embedded User Flow


1
Create Chain (.xcf) with Programmer
2
Generate VME Using Deployment Tool
3
Test Using Download Debugger
4
Modify Eprom-based Source Code
5
Compile Source Code w/ Hex Files
6
Modify source code file hardware.c
7
Compile source code and VME HEX files

Step 3. Convert VME to HEX

A HEX file can be created from a VME file by using the vme2hex source code that is shipped with Programmer, or by selecting the Generate HEX (.c) File option in Deployment Tool. This source code can be found in the installation path of Programmer, under the <*install_path*>\embedded_source\vmembedded\sourcecode\svf2vme. A HEX file is a C-programming language file that has the VME byte codes converted and stored in an array.

Step 4. Modify EPROM-based Source Code

The file hardware.c must be modified to target the embedded system. In particular, the following functions must be changed to be able to write, read, and observe the delay, respectively:

- `void writePort( unsigned char a_ucPins, unsigned char a_ucValue )`
- `unsigned char readPort()`
- `void ispVMDelay(unsigned int delay_time)`

Step 5. Compile Source Code and HEX Files

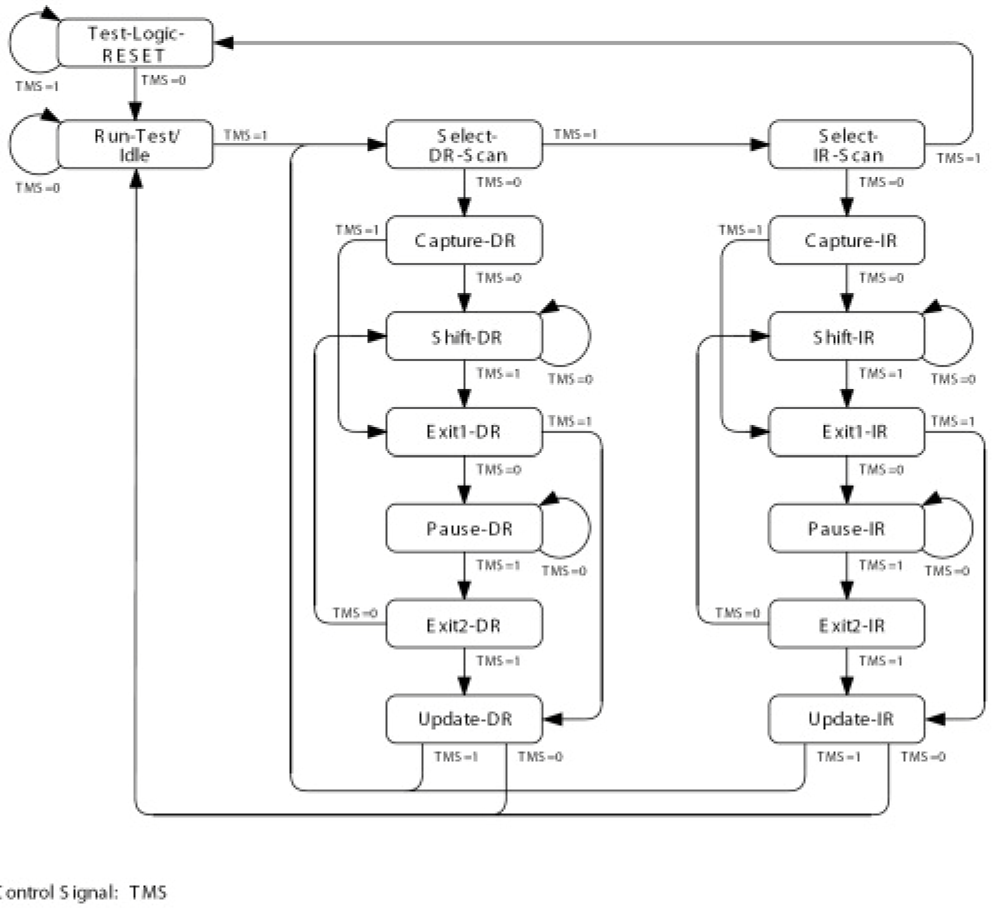
Combine the source code and HEX files into a project to be compiled. This may be done by using a microcontroller compiler.

Programming Engine Flow

The programming engine of the JTAG Full VME Embedded is driven by the byte codes of the VME file. The byte codes instruct the programming engine as to what sequence of functions to follow in order to shift in instructions, move the TAP controller state machine, shift data in and out of the device, and observe delays.

The TAP controller is a synchronous state machine that is based on the TMS (Test Mode Select) and TCK (Test Clock) signals of the TAP and controls the sequence of operations of the circuitry defined by the IEEE 1149.1 standard. The TCK signal can be driven at a maximum of 25 MHz. JTAG devices in the chain may limit the TCK speed. Confirm the Maximum TCK for all the devices in the programming chain.

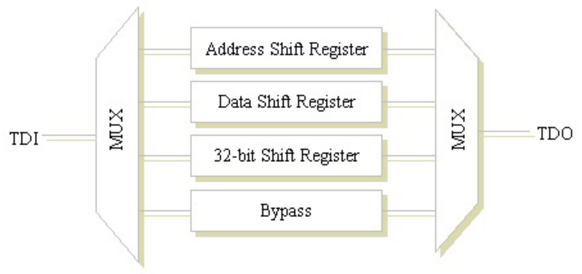
Figure 15: TAP Controller State Diagram



In the Shift-DR state, a decoder is present to select which shift register is enabled and connects it between TDI and TDO. The following are the shift registers: Address Shift Register, Data Shift Register, 32-bit Shift Register, and Bypass. The 32-bit Shift Register is used to store the ID code and

USERCODE. The first bit clocked into any of the registers is placed into the MSB, and data is shifted towards TDO as additional bits are clocked in.

Figure 16: Shift Registers



The engine core is implemented as a switch statement. The cases in this switch statement perform specialized functions based on the byte code and its operand(s). These functions may end up calling other switch statements, calling the engine core recursively, setting global variable values, or interfacing with the device directly. Once the byte code instruction has been executed, it returns to the main switch statement to process the next byte.

The processor begins by calculating the 16-bit CRC of the VME file and comparing it against the expected CRC. If that is successful, the processor then verifies the version of the VME file to make sure it is supportable. The version is an eight byte ASCII of the format ____`<Major Version>.<Minor Version>`, where `<Major Version>` and `<Minor Version>` are digits `0-9`. If the version verification fails, the processor returns the error code –4 to indicate a file error.

The Main Engine Switch calls the appropriate case statements based on the incoming byte code from the VME. Unrecognized byte codes will result in the program exiting with the error code –4 to indicate a file error.

STATE Case Statement

The STATE case statement expects a state following the STATE byte code to instruct the processor to step the IEEE 1149.1 bus to the next state. The state must be a valid stable state, which is IRPAUSE, DRPAUSE, RESET, or IDLE

SIR Case Statement

The SIR case statement begins by extracting the size of the register. The size will be used later to indicate how many bits of data will be sent or read back from the device. If the flow control has been set to CASCADE, then the processor shifts the device to the SHIFTIR. The presence of CASCADE in the flow control indicates that the SIR instruction is targeting over 64Kb of data and has been broken down to ease the memory requirements.

Figure 17: Main Engine Switch

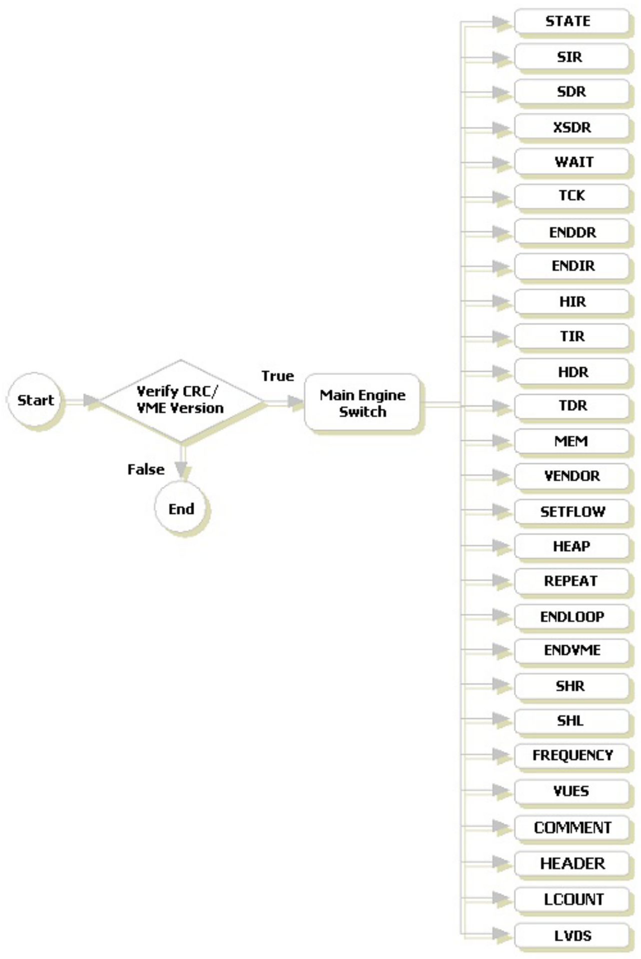
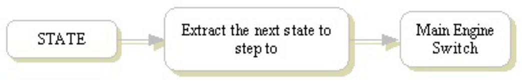


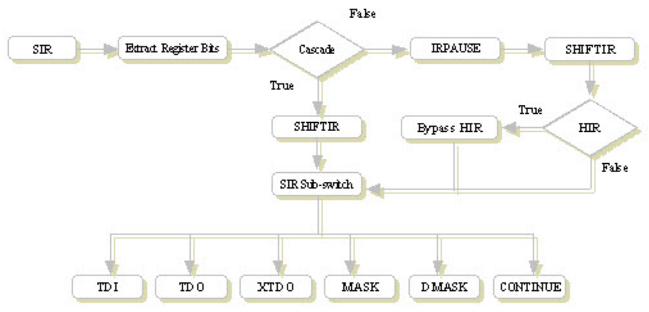
Figure 18: STATE Case Statement



If CASCADE has not been set, then the processor shifts the device into the safe state IRPAUSE, and then to SHIFTIR. If HIR exists (see HIR Case Statement), then the processor will bypass the HIR. The SIR sub-switch is a switch that is based off of the byte codes that can potentially be found after the SIR byte code.

The TDI byte code indicates that there is data that needs to be shifted into the device. The data following the TDI byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be shifted into the device.

Figure 19: SIR Case Statement



The TDO byte code indicates that there is data that needs to be read and verified from the device. The data following the TDO byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be compared against data in the device.

The XTDO byte code indicates that the TDO data is the TDI data of the previous frame, such as in the case of concurrent, or turbo, programming. Data will not follow the XTDO byte code, resulting in a smaller VME. Instead, the previous frame’s TDI data will be used as the current TDO data.

The MASK byte code indicates that there is mask data that needs to be used when comparing the TDO values against the actual values read from the device. The data following the MASK byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be used when comparing against data in the device.

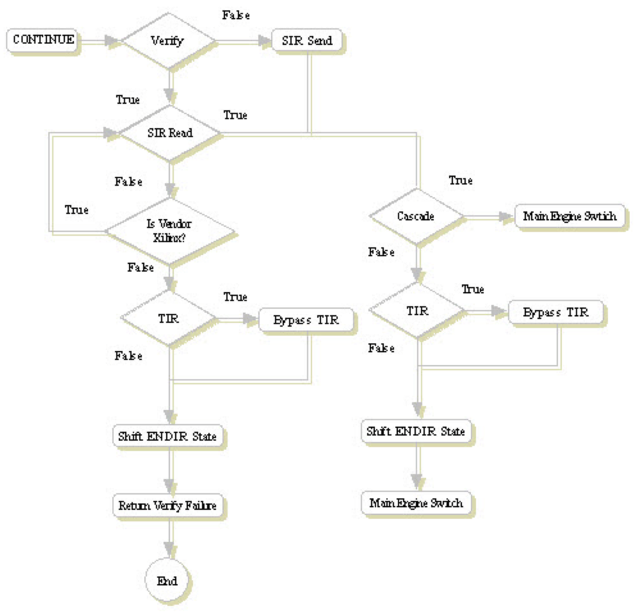
The DMASK byte code indicates that there is dynamic mask data that needs to be used when comparing the boundary scan. The data following the DMASK byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be used when comparing against the boundary scan.

The CONTINUE byte code terminates the SIR instruction. When this byte is encountered, it indicates that the processor is ready to send or read and verify data from the device using the data it is currently holding in memory. If any byte codes other than TDI, TDO, XTDO, MASK, DMASK, and CONTINUE were encountered in the SIR Sub-switch, the program will exit with the error code –4, indicating a file error.

If the TDO or XTDO byte code were encountered in the SIR sub-switch, then that indicates that the SIR instruction is going to read data from the device, else the SIR instruction is going to send data to the device.

If reading and verifying data from the device were successful, the processor checks if the CASCADE flag has been set. If it is set, the control returns to the Main Engine Switch. If the flag is off, the processor checks if TIR exists (see TIR Case Statement). If it exists, then the trailer devices must be bypassed. Next, it shifts the device to the stable state that followed the ENDIR byte code

Figure 20: SIR Case Statement Continued



(see ENDIR Case Statement). The control returns back to the Main Engine Switch.

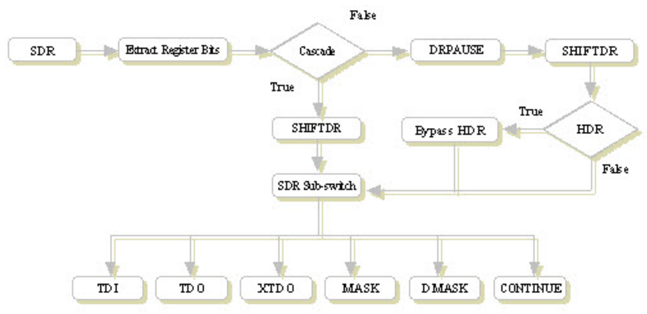
If reading and verifying data from the device were unsuccessful, the processor checks if the vendor has been set to Xilinx. If the vendor is Xilinx, repeat the read loop up to 30 times before returning an error. If the vendor is not Xilinx, the processor bypasses the TIR if there are trailer devices. Next, it shifts the device to the stable state that followed the ENDIR byte code. The error code returned is –1 to indicate a verification failure.

If TDO or XTDO were not encountered in the SIR sub-switch, then the processor sends data to the device. If the CASCADE flag has been set, the control returns to the Main Engine Switch. If TIR exists, then the trailer devices must be bypassed. Next, it shifts the device to the stable state that followed the ENDIR byte code. The control returns back to the Main Engine Switch.

SDR Case Statement

The SDR case statement works similar to the SIR. It begins by extracting the size of the register. The size will be used later to indicate how many bits of data will be sent or read back from the device. If the flow control has been set to CASCADE, then the processor shifts the device to the SHIFTIR. The presence of CASCADE in the flow control indicates that the SDR instruction is targeting over 64Kb of data and has been broken down to ease the memory requirements.

Figure 21: SDR Case Statement



If CASCADE has not been set, then the processor shifts the device into the safe state DRPAUSE, and then to SHIFTDR. If HDR exists (see HDR Case Statement), then the processor will bypass the HDR. The SDR sub-switch is a switch that is based off the byte codes that can potentially be found after the SDR byte code.

The TDI byte code indicates that there is data that needs to be shifted into the device. The data following the TDI byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be shifted into the device.

The TDO byte code indicates that there is data that needs to be read and verified from the device. The data following the TDO byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be compared against data in the device.

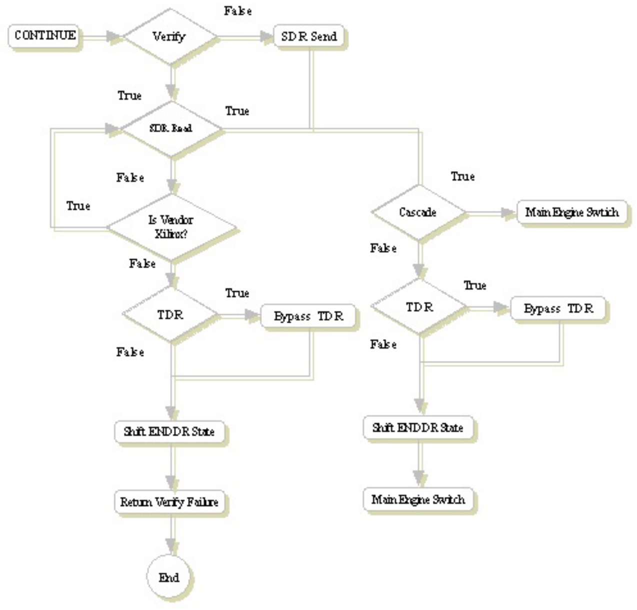
he XTDO byte code indicates that the TDO data is the TDI data of the previous frame, such as in the case of concurrent, or turbo, programming. Data will not follow the XTDO byte code, resulting in a smaller VME. Instead, the previous frame's TDI data will be used as the current TDO data.

The MASK byte code indicates that there is mask data that needs to be used when comparing the TDO values against the actual values read from the device. The data following the MASK byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be used when comparing against data in the device.

The DMASK byte code indicates that there is dynamic mask data that needs to be used when comparing the boundary scan. The data following the DMASK byte will be extracted and decompressed, if compression were selected, and held in memory until it is ready to be used when comparing against the boundary scan.

The CONTINUE byte code terminates the SDR instruction. When this byte is encountered, it indicates that the processor is ready to send or read and verify data from the device using the data it is currently holding in memory. If any byte codes other than TDI, TDO, XTDO, MASK, DMASK, and CONTINUE were encountered in the SDR Sub-switch, the program will exit with the error code –4, indicating a file error.

Figure 22: SDR Case Statement Continued



If the TDO or XTDO byte code were encountered in the SDR sub-switch, then that indicates that the SDR instruction is going to read data from the device, else the SDR instruction is going to send data to the device.

If reading and verifying data from the device were successful, the processor checks if the CASCADE flag has been set. If it is set, the control returns to the Main Engine Switch. If the flag is off, the processor checks if TDR exists (see TDR Case Statement). If it exists, then the trailer devices must be bypassed. Next, it shifts the device to the stable state that followed the ENDDR byte code (see ENDDR Case Statement). The control returns back to the Main Engine Switch.

If reading and verifying data from the device were unsuccessful, the processor checks if the vendor has been set to Xilinx. If the vendor is Xilinx, repeat the read loop up to 30 times before returning an error. If the vendor is not Xilinx, the processor bypasses the TIR if there are trailer devices. Next, it shifts the device to the stable state that followed the ENDIR byte code. The error code returned is –1 to indicate a verification failure.

If TDO or XTDO were not encountered in the SDR sub-switch, then the processor sends data to the device. If the CASCADE flag has been set, the control returns to the Main Engine Switch. If TDR exists, then the trailer devices must be bypassed. Next, it shifts the device to the stable state that followed the ENDDR byte code. The control returns back to the Main Engine Switch.

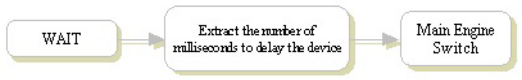
XSDR Case Statement

The XSDR case statement works exactly like the SDR case statement, except that it sets the EXPRESS flag. The EXPRESS flag indicates to the processor that the VME is performing concurrent programming. Therefore, the TDO data shall use the previous frame's TDI data. This reduces the VME size drastically because the data is not duplicated.

WAIT Case Statement

The WAIT case statement expects a number following the WAIT byte code to represent the number of milliseconds of delay the device must observe. The delay is observed immediately. The user must update the delay function in the source code to make the target embedded system observe the delay duration correctly.

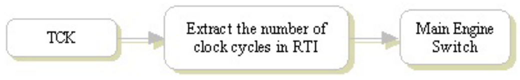
Figure 23: WAIT Case Statement



TCK Case Statement

The TCK case statement expects a number following the TCK byte code to represent the number of clocks that the device must remain in the run test idle state. The clock cycles are executed immediately into the device.

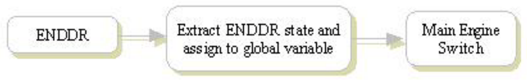
Figure 24: TCK Case Statement



ENDDR Case Statement

The ENDDR case statement expects a state following the ENDDR byte code to represent the TAP state that the processor shall move the device to after an SDR instruction. This state will be stored in a global variable.

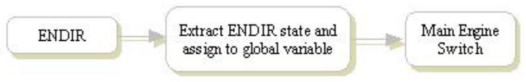
Figure 25: ENDDR Case Statement



ENDIR Case Statement

The ENDIR case statement expects a state following the ENDIR byte code to represent the TAP state that the processor shall move the device to after an SIR instruction. This state will be stored in a global variable.

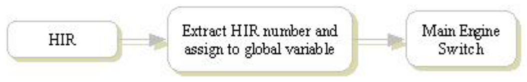
Figure 26: ENDIR Case Statement



HIR Case Statement

The HIR case statement expects a number following the HIR byte code to represent the number of header devices. The number will be stored in a global variable and the processor will issue this number of bypasses prior to executing an SIR instruction.

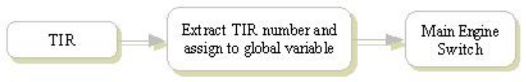
Figure 27: HIR Case Statement



TIR Case Statement

The TIR case statement expects a number following the TIR byte code to represent the number of trailer devices. The number will be stored in a global variable and the processor will issue this number of bypasses after executing an SIR instruction.

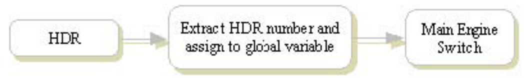
Figure 28: TIR Case Statement



HDR Case Statement

The HDR case statement expects a number following the HDR byte code to represent the number of header register bits. The number will be stored in a global variable and the processor will issue this number of bypasses prior to executing an SDR instruction.

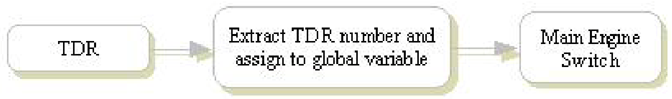
Figure 29: HDR Case Statement



TDR Case Statement

The TDR case statement expects a number following the TDR byte code to represent the number of trailer register bits. The number will be stored in a global variable and the processor will issue this number of bypasses after executing an SDR instruction.

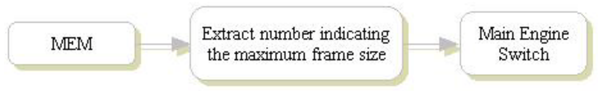
Figure 30: TDR Case Statement



MEM Case Statement

The MEM case statement expects a number following the MEM byte code to represent the maximum frame size in bits. Memory buffers will be allocated for TDI, TDO, MASK, and DMASK data according to the maximum number.

Figure 31: MEM Case Statement

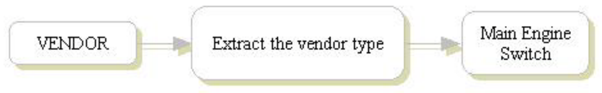


VENDOR Case Statement

The VENDOR case statement expects the vendor type following the VENDOR byte code to represent the vendor the VME supports. Different vendors require different programming algorithms that must be supported.

This byte notifies the embedded processor to enable the specified vendor support.

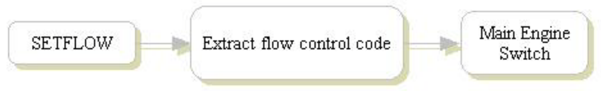
Figure 32: VENDOR Case Statement



SETFLOW Case Statement

The SETFLOW case statement expects an instruction following the SETFLOW byte code to instruct the embedded processor to enable certain properties during execution. This is useful for cascading and looping VME files, where the processor flow must change in order to take advantage of these features.

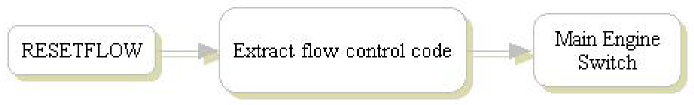
Figure 33: SETFLOW Case Statement



RESETFLOW Case Statement

The RESETFLOW case statement works to reset the properties enabled during the SETFLOW case statement.

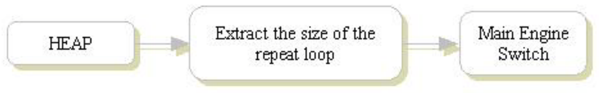
Figure 34: RESETFLOW Case Statement



HEAP Case Statement

The HEAP case statement expects a number following the HEAP byte code to indicate the size of the upcoming repeat loop. In the file-based JTAG Full VME Embedded, this size is used to dynamically allocate memory to hold the repeat loop. In the EPROM-based embedded, the heap array is set to point to the heap buffer in the HEX file.

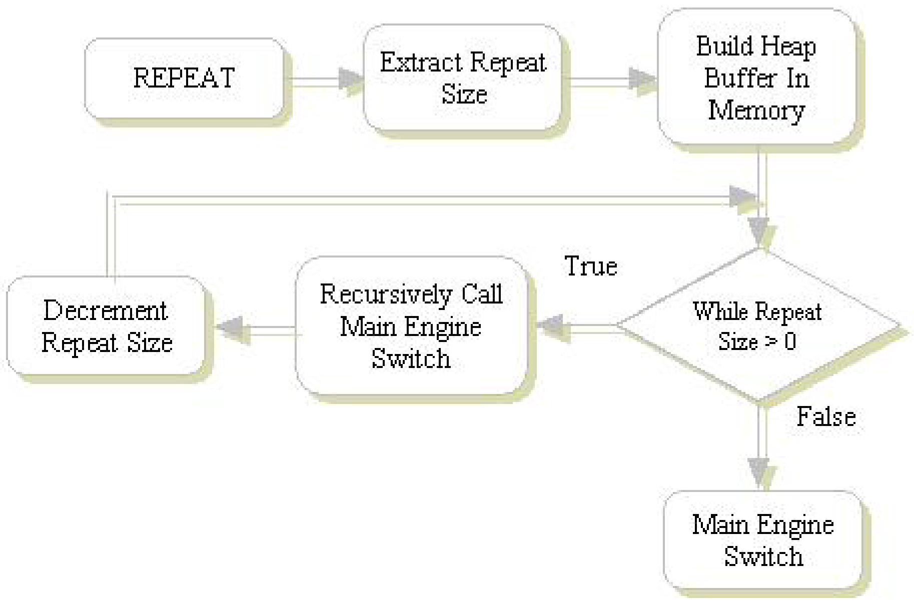
Figure 35: HEAP Case Statement



REPEAT Case Statement

The REPEAT case statement is executed if the VME were generated with the looping option. A looping VME attempts to reduce the VME size by forming loops around similar algorithm. Following the REPEAT byte code, a number indicating the number of repeats is extracted. The heap buffer is build by reading the number of HEAP size (see HEAP case statement) bytes and storing them in memory. Recursive calls are made back to the Main Engine Switch, which will process the byte codes within the heap buffer. The recursive calls end when the repeat size is zero.

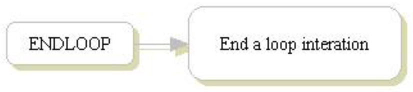
Figure 36: REPEAT Case Statement



ENDLOOP Case Statement

The ENDLOOP byte code terminates a loop iteration and shall be encountered only when the embedded processor is processing a repeat loop. This byte code shall always be the last byte of the heap buffer. When this byte code is found, the control returns back to the looping control, where the repeat size gets decremented and the next iteration of the loop begins, unless the repeat size is zero.

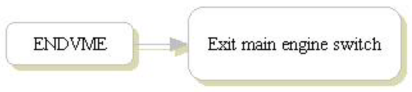
Figure 37: ENDLOOP Case Statement



ENDVME Case Statement

The ENDVME case statement exits the main engine switch. This byte code is the last byte of the VME.

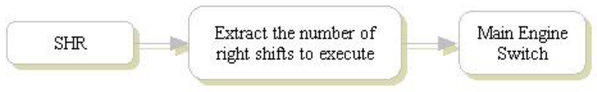
Figure 38: ENDVME Case Statement



SHR Case Statement

The SHR case statement expects a number following the SHR byte code to perform a right shift on the TDI data buffer. At this point the TDI data buffer should store the register address. By simply right shifting the register address to increment to the next frame instead of having the VME contain several register address buffers, the VME size is reduced.

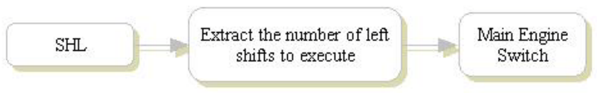
Figure 39: SHR Case Statement



SHL Case Statement

The SHL case statement works similar to the SHR case statement, except that it shifts to the left.

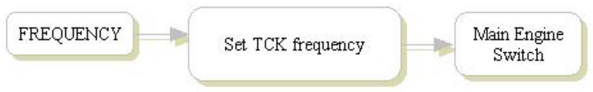
Figure 40: SHL Case Statement



FREQUENCY Case Statement

The FREQUENCY case statement expects a number following the FREQUENCY byte code to establish the TCK frequency.

Figure 41: FREQUENCY Case Statement



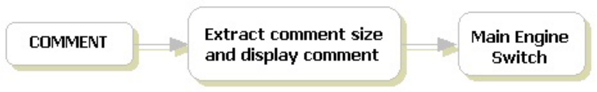
VUES Case Statement

The VUES case statement sets the flow control register to indicate that the VME is invoking the Continue If Fail feature. Under this condition, if the USERCODE verification fails, then the embedded processor continues with programming the data. If the USERCODE verification passes, then the processor exits without programming.

COMMENT Case Statement

The COMMENT case statement is executed if the VME file were generated to support SVF comments. This statement expects a number to indicate the size of the comment. The comment is then read one byte at a time and displayed onto the terminal. It ends when the number of bytes processed equals the number indicating the size of the comment.

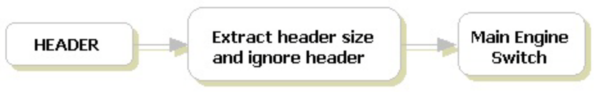
Figure 42: COMMENT Case Statement



HEADER Case Statement

The HEADER case Statement is executed if the VME file were generated with header information. Currently, this feature is not supported.

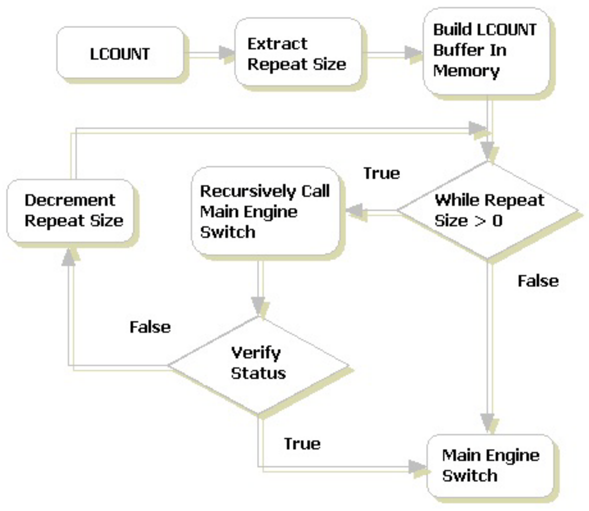
Figure 43: HEADER Case Statement



LCOUNT Case Statement

The LCOUNT case statement is executed if the VME file targets FLASH or PROM devices. It allows the engine to repeatedly check the status of the device before programming the next block of data. This statement expects a number to indicate the number of status checks before issuing a failure return code. The engine will use an index to point to the repeated commands in a buffer and issue them to the device. The index is reset after each iteration. This will continue until the number of status checks gets decremented to zero, or until the status is verified to be true.

Figure 44: LCOUNT Case Statement



LVDS Case Statement

The LVDS case statement informs the processor about the number of LVDS pairs and which are paired. This ensures that the processor will drive opposite values back into the pairs.

VME Byte Codes

Appendix C lists the byte codes that are found in the VME and interpreted by the embedded processor.

General Opcode	Value	Description
VMEHEXMAX	60000L	Sets the HEX file maximum size to 60K
SCANMAX	64000L	Sets the maximum data burst to 64K

Formatting Opcode	Value	Description
CONTINUE	0x70	Indicates the end of a VME line
ENDVME	0x7F	Indicates the end of a VME file
ENDFILE	0xFF	Indicates the end of file

JTAG Opcode	Value	Description
RESET	0x00	Traverse to TLR
IDLE	0x01	Traverse to RTI
IRPAUSE	0x02	Traverse to PAUSE IR
DRPAUSE	0x03	Traverse to PAUSE DR
SHIFTIR	0x04	Traverse to SHIFT IR
SHIFTDR	0x05	Traverse to SHIFT DR

Flow Control Opcode	Value	Description
INTEL_PRGM	0x0001	Intelligent programming in effect
CASCADE	0x0002	Currently splitting large SDR
REPEATLOOP	0x0008	Currently executing a repeat loop
SHIFTRIGHT	0x0080	Indicates the next stream needs a right shift
SHIFTLEFT	0x0100	Indicates the next stream needs a left shift
VERIFYUES	0x0200	Indicates Continue If Fail flag

Data Type Register Opcode	Value	Description
EXPRESS	0x0001	Simultaneous program and verify
SIR_DATA	0x0002	SIR is the active SVF command
SDR_DATA	0x0004	SDR is the active SVF command
COMPRESS	0x0008	Data is compressed
TDI_DATA	0x0010	TDI data is present
TDO_DATA	0x0020	TDO data is present
MASK_DATA	0x0040	MASK data is present
HEAP_IN	0x0080	Data is from the heap
LHEAP_IN	0x0200	Data is from the intelligent data buffer
VARIABLE	0x0400	Data is from a declared variable
CRC_DATA	0x0800	CRC data is present
CMASK_DATA	0x1000	CMASK data is present
RMASK_DATA	0x2000	RMASK data is present
READ_DATA	0x4000	READ data is present
DMASK_DATA	0x8000	DMASK data is present

Hardware Opcode	Value	Description
signalENABLE	0x1C	Assert the ispEN pin
signalTMS	0x1D	Assert the MODE or TMS pin
signalTCK	0x1E	Assert the SCLK or TCK pin
signalTDI	0x1F	Assert the SDI or TDI pin
signalTRST	0x20	Assert the RESTE or TRST pin

Vendor Opcode	Value	Description	
VENDOR	0x56	Indicates vendor opcode is following	
LATTICE	0x01	Indicates Lattice or JTAG device	
ALTERA	0x02	Indicates Altera device	

XILINX	0x03	Indicates Xilinx device	

SVF Opcode	Value	Description
ENDDATA	0x00	Indicates the end of the current SDR data stream
RUNTEST	0x01	Indicates the duration to stay at the stable state
ENDDR	0x02	Indicates the stable state after SDR
ENDIR	0x03	Indicates the stable state after SIR
ENDSTATE	0x04	Indicates the stable state after RUNTEST
TRST	0x05	Assert the TRST pin
HIR	0x06	Specifies the sum of IR bits at lead
TIR	0x07	Specifies the sum of IR bits at end
HDR	0x08	Specifies the number of devices at lead
TDR	0x09	Specifies the number of devices at end
ispEN	0x0A	Assert the ispEN pin
FREQUENCY	0x0B	Specifies the maximum clock rate to run the state machine
STATE	0x10	Move to the next stable state
SIR	0x11	Indicates the instruction stream is following
SDR	0x12	Indicates the data stream is following
TDI	0x13	Indicates the data stream following is fed into the device
TDO	0x14	Indicates the data stream is to be read and compare
MASK	0x15	Indicates the data stream following is the output mask
XSDR	0x16	Indicates the data stream following is for simultaneous shift in and shift out
XTDI	0x17	Indicates the data stream following is for shift in only and it must be stored for verifying on the next XSDR call
XTDO	0x18	Indicates there is no data stream following, instead it should be retrieved from the previous XTDI token
MEM	0x19	Indicates the size of the memory needed to be allocated.
WAIT	0x1A	Indicates the duration of the delay at IDLE state
TCK	0x1B	Indicates the number of clocks to pulse to TCK
HEAP	0x32	Indicates the size of the memory needed to hole the loop
REPEAT	0x33	Indicates the beginning of a reap loop
LEFTPAREN	0x35	Indicates the beginning of the data following the loop
VAR	0x55	Indicates a place holder for the data if looping option has been selected
SEC	0x1C	Indicates the absolute time in seconds that must be observed
SMASK	0x1D	Indicates the mask for TDI data
MAX	0x1E	Indicates the absolute maximum wait time
ON	0x1F	Assert the targeted pin
OFF	0x20	Dis-assert the targeted pin
SETFLOW	0x30	Change the Flow Control Register
RESETFLOW	0x31	Clear the Flow Control Register
CRC	0x47	Indicates which bits may be used in CRC calculation
CMASK	0x48	Indicates which bits shall be used in CRC calculation
RMASK	0x49	Indicates which bits shall be used in Read and Save
READ	0x50	Indicates which bits may be used in Read and Save
ENDLOOP	0x59	Indicates the end of the repeat loop
SECUREHEAP	0x60	Byte encoded to secure the HEAP structure

SVF Opcode	Value	Description
VUES	0x61	Indicates Continue If Fail option has been selected
DMASK	0x62	Indicates SVF file has DMASK
COMMENT	0x63	Support SVF comments in VME file
HEADER	0x64	Support header in VME file
FILE_CRC	0x65	Support CRC-protected VME file
LCOUNT	0x66	Support intelligent programming.
LDELAY	0x67	Support intelligent programming.
LSDR	0x68	Support intelligent programming.
LHEAP	0x69	Memory needed to hold intelligent data buffer
LVDS	0x71	Support LVDS

Return Codes	Value	Description
VME_VERIFICATION_ERROR	-1	Value returned when the expected data does not match the actual data of the device
VME_FILE_READ_FAILURE	-2	Value returned when the VME file cannot be read
VME_VERSION_FAILURE	-3	Value returned when the version is not supported
VME_INVALID_FILE	-4	Value returned when an invalid opcode is encountered
VME_ARGUMENT_FAILURE	-5	Value returned when a command line argument is invalid
VME_CRC_FAILURE	-6	Value returned when the expected CRC does not match the calculated CRC.

Unsupported SVF Syntax

The following are the SVF syntax not supported by the SVF2VME utility:

- TRST - The TRST command is ignored.
- PIO - The PIO command will cause SVF2VME to exit with an error.
- PIOMAP - The PIOMAP command will cause SVF2VME to exit with an error.
- MAXIMUM - The optional parameter MAXIMUM is ignored. This may be found in the RUNTEST command.
- SMASK - The optional parameter SMASK is ignored. This may be found in the HDR, HIR, TDR, TIR, SIR, or SDR commands.
- Explicit state transitions in the STATE command that contain non-stable states will cause SVF2VME to exit with an error. Only transitions between stable states are supported in the table below.
- STATE RESET is supported. However, it is strongly discouraged to be included into the SVF file. This statement causes the undesirable effect of having all the devices in the entire JTAG chain to be reseted.

The following table indicates the paths taken between stable states.

Current State	New State	State Path
RESET	RESET	RESET (NO CLOCK)
RESET	IDLE	RESET-IDLE
RESET	DRPAUSE	RESET-IDLE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
RESET	IRPAUSE	RESET-IDLE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
IDLE	RESET	IDLE-DRSELECT-IRSELECT-RESET
IDLE	IDLE	IDLE (NO CLOCK)
IDLE	DRPAUSE	IDLE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
IDLE	IRPAUSE	IDLE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
DRPAUSE	RESET	DRPAUSE-DREXIT2-DRUPDATE-DRSELECT-IRSELECT-RESET
DRPAUSE	IDLE	DRPAUSE-DREXIT2-DRUPDATE-IDLE
DRPAUSE	DRPAUSE	DRPAUSE (NO CLOCK)
DRPAUSE	IRPAUSE	DRPAUSE-DREXIT2-DRUPDATE-DRSELECT-IRSELECT-IRCAPTURE-IREXIT1-IRPAUSE
IRPAUSE	RESET	IRPAUSE-IREXIT2-IRUPDATE-DRSELECT-IRSELECT-RESET
IRPAUSE	IDLE	IRPAUSE-IREXIT2-IRUPDATE-IDLE
IRPAUSE	DRPAUSE	IRPAUSE-IREXIT2-IRUPDATE-DRSELECT-DRCAPTURE-DREXIT1-DRPAUSE
IRPAUSE	IRPAUSE	IRPAUSE (NO CLOCK)

JTAG Slim VME Embedded

The JTAG Slim VME Embedded software, based on the serial vector format file, enables you to quickly and efficiently program chains of devices, thus improving productivity and lowering costs. The JTAG Slim VME Embedded code is designed for microcontrollers with limited resources, such as the 8051 microcontroller.

The JTAG Slim VME Embedded software behaves the same as the JTAG Full VME Embedded. The difference is it is geared to a 8051 processor. The C code adds memory space keywords specific to the 8051 processor. The size of the devices which can be programmed are limited by the amount of contiguous SRAM available to the 8051 processor.

The JTAG Slim VME Embedded is available with installations of Radiant Programmer. Its advantages over other embedded systems include:

- **Footprint of less than 3KB ROM** – The small footprint is made possible by optimizing the JTAG Slim VME Embedded programming engine to use the least amount of code in the most efficient fashion.
- **Reduced RAM usage** – The memory usage is fixed at a minimal set for all IEEE 1532-compliant devices. The number of global and local variables has been reduced to a minimum, and no data buffers are required to be held in memory.
- **Compressible algorithm and data** – The programming data, calculated by multiplying the frame size by the number of frames, can increase the ROM requirement substantially. For example, the device LC51024MV(B) has a frame size of 2624 with 388 frames, resulting in 125 KB of ROM. Fortunately, the JTAG Slim VME Embedded can compress the programming data into sizes that are much smaller. The compression is performed frame by frame and is data file dependent.

- **Sequential chain programming** – The JTAG Slim VME Embedded can process multiple devices in the same chain, with mixed operations in sequential mode.

See Also ▶Using the PC-based JTAG Slim VME Embedded

- Using the 8051-based JTAG Slim VME Embedded

JTAG Slim VME Embedded Source Code

The source code for both the PC-based and the 8051-based JTAG Slim VME Embedded can be found in the <*install_path*>\embedded_source\slimembedded directory.

Each project has the following files. The major entry point for JTAG Slim VME Embedded is slim_vme.c.

slim_vme.c The slim_vme.c file is the only file that differs between the PC-based and the 8051-based embedded solutions. This difference is due to the way each of these interfaces to the VME algorithm and data files through the entry point. This file contains the main and entry point functions.

slim_pro.c The slim_pro.c file provides the programming engine for the JTAG Slim VME Embedded. The engine operates on the commands in the VME algorithm, and fetches data from the VME data, if necessary. The engine is responsible for functions such as sending data, verifying data, observing timing delay, stepping through the state machine, decompression, and so on.

hardware.c The only file that you should modify is hardware.c. This file contains the functions to read and write to the port and the timing delay function. You must update these functions to target the desired hardware. The released version targets the parallel port of the PC at address 0x0378 using Lattice's download cable.

opcode.h The opcode.h file contains the definitions of the byte codes used in the VME algorithm format and programming engine.

debug.h The debug.h file will print out debugging information if the preprocessor switch VME_DEBUG is defined in the project. This is an optional file to include.

windriver.c and windriver.h The windriver.c and windriver.h files target the JTAG Slim VME Embedded to Windows. These files will be compiled if the preprocessor switch VME_WINDOWS is defined in the project file. These files should be omitted when compiling the 8051-based JTAG Slim VME Embedded onto an embedded platform.

See Also ▶VME Algorithm Format

- VME Data Format

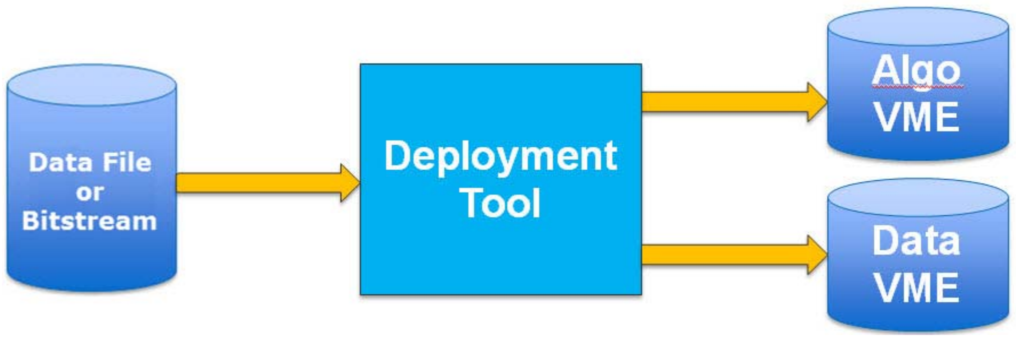
Using the PC-based JTAG Slim VME Embedded

The PC-based JTAG Slim VME Embedded is a quick and easy way to validate the VME files and the JTAG Slim VME Embedded programming engine by successfully processing the target chain of IEEE 1532 compliant devices using the parallel port of the PC.

The JTAG Slim VME Embedded system uses a compressed binary variation of SVF files, called VME, as input. Like the SVF file, the VME file contains high-level IEEE 1149.1 bus operations. These operations consist of scan operations and movements between the IEEE 1149.1 TAP states. However, unlike the SVF file, where the programming algorithm of the device is intermeshed with the programming data, the VME file is separated into a VME algorithm file and a VME data file. This separation of the algorithm and data allows the optimization of the JTAG Slim VME Embedded programming engine. It also allows you to mix VME data files with VME algorithm files, provided the chain and operations are the same.

Figure 9 shows an example of Slim VME embedded file generation for the JTAG port.

Figure 45: Slim VME Embedded VME Flow



The JTAG Slim VME Embedded capability is enabled only if all the following conditions are met:

- All the devices in the chain are IEEE-1532 compliant.
- Sequential mode is selected.
- Synchronize Enable and Disable setting is unchecked.
- Operation is not Read and Save or a display operation such as Calculate Checksum or Display ID.

See Also ▶Generating JTAG Slim VME Embedded Files

Using the 8051-based JTAG Slim VME Embedded

To program embedded systems using the 8051-based JTAG Slim VME Embedded, you must generate the VME files as HEX to create the VME algorithm and data files as C programming files. Each file contains a C programming style byte buffer that holds the VME algorithm or data.

The HEX files must be compiled along with the 8051-based JTAG Slim VME Embedded source code. The source code contains handles that allow the compiler to link the buffers of the hexadecimal files to the main source code. The only source code file that you need to modify is the hardware.c file. You must implement methods to write and read to the hardware port, as well as observe the timing delay. You must modify the following functions according to the target platform:

- readPort
- writePort
- ispVMDelay

The following are optional functions that you may wish to modify in the hardware.c file in order to enable and disable the hardware conditions before and after processing:

- EnableHardware
- DisableHardware

See Also ▶Generating JTAG Slim VME Embedded Files

- JTAG Slim VME Embedded Source Code

VME Algorithm Format

The VME algorithm file contains byte codes that represent the programming algorithm of the device or chain.

VME Symbol	HEX Value
STATE	0x01
SIR	0x02
SDR	0x03
TCK	0x04
WAIT	0x05
ENDDR	0x06
ENDIR	0x07
HIR	0x08

VME Symbol	HEX Value
TIR	0x09
HDR	0x0A
TDR	0x0B
BEGIN_REPEAT	0x0C
FREQUENCY	0x0D
TDI	0x0E
CONTINUE	0x0F
END_FRAME	0x10
TDO	0x11
MASK	0x12
END_REPEAT	0x13
DATA	0x14
PROGRAM	0x15
VERIFY	0x16
ENDVME	0x17
DTDI	0x18
DTDO	0x19

The byte codes perform the same operations as the SVF commands, with the exception of BEGIN_REPEAT, CONTINUE, END_FRAME, END_REPEAT, DATA, PROGRAM, VERIFY, ENDVME, DTDI, and DTDO.

The byte codes BEGIN_REPEAT, END_REPEAT, PROGRAM, VERIFY, DTDI, and DTDO are used to support a repeating VME algorithm structure to minimize the algorithm size, a feature that the linear SVF does not provide.

The byte code CONTINUE appears at the end of every SIR and SDR instruction as a terminator.

The byte code END_FRAME appears at the end of every frame in the VME data as a terminator.

Translation from the SVF file to VME algorithm file is done command by command. For example, the following SVF line:

```
SIR 8 TDI (16);
```

will be converted to the following VME line, in binary:

```
0x02 0x08 0x0E 0x68
```

The VME Algorithm file is similar to the SVF file with the following differences:

- VME Algorithm uses byte codes from the table below to represent SVF commands
- Fuse data and USERCODE have been removed
- Looping algorithm

The following is an example of an EPV VME Algorithm file and the SVF translation for the LC4064V device:

Table 2: VME Algorithm Example

VME Algorithm Format	Serial Vector Format (SVF)	Description
0x0A 0x00	HDR 0;	
0x08 0x00	HIR 0;	
0x0B 0x00	TDR 0;	
0x09 0x00	TIR 0;	
0x06 0x03	ENDDR DRPAUSE;	
0x07 0x02	ENDIR IRPAUSE;	
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0x68 0x0F	SIR 8 TDI (16);	Shift in the IDCODE instruction
0x01 0x01	STATE IDLE;	
0x03 0x20 0x0E 0xFF 0xFF 0xFF 0xFF 0x11 0xC2 0x09 0x01 0x80 0x12 0xFF 0xFF 0xF0 0x0F	SDR 32 TDI (FFFFFFFF) TDO (01809043) MASK (0FFFFFFF);	Verify the IDCODE
0x02 0x08 0x0E 0x38 0x0F	SIR 8 TDI (1C);	Shift in the PRELOAD instruction
0x03 0x44 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x0F	SDR 68 TDI (000000000000000000);	Shift all zero data into boundary scan cells
0x02 0x08 0x0E 0xA8 0x0F	SIR 8 TDI (15);	Shift in ENABLE instruction
0x01 0x01 0x04 0x03 0x05 0x14 0x01 0x01	RUNTEST IDLE 3 TCK 2.00E-002 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0xC0 0x0F	SIR 8 TDI (03);	Shift in ERASE instruction
0x01 0x01 0x04 0x03 0x05 0x64 0x01 0x01	RUNTEST IDLE 3 TCK 1.00E-001 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0x84 0x0F	SIR 8 TDI (21);	Shift in ADDRESS INIT instruction
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0xE4 0x0F	SIR 8 TDI (27);	Shift in PROGRAM INCR instruction
0x0C 0x5F 0x15	N/A	Begin PROGRAM repeat loop of size 95
0x03 0xE0 0x02 0x18 0x14 0x0F	SDR 352 DTDI (DATA);	Notice the forth byte is 0x18, which is actually DTDI. DTDI instructs the processor to send in data from the data buffer
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x13	N/A	Terminate the repeat algorithm
0x02 0x08 0x0E 0x58 0x0F	SIR 8 TDI (1A);	Shift in PROGRAM USERCODE instruction
0x03 0x20 0x18 0x14 0x0F	SDR 32 DTDI (DATA);	Shift in the USERCODE The USERCODE can be found in the data buffer.
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0x80 0x0F	SIR 8 TDI (01);	Shift in ADDRESS SHIFT instruction

Table 2: VME Algorithm Example (Continued)

0x03 0x5F 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x0F	SDR 95 TDI (400000000000000000000000);	Shift in beginning address
0x02 0x08 0x0E 0x54 0x0F	SIR 8 TDI (2A);	Shift in READ INC instruction
0x0C 0x5F 0x16	N/A	Begin VERIFY repeat loop of size 95
0x01 0x01 0x04 0x03 0x05 0x01 0x01 0x01	RUNTEST IDLE 3 TCK 1.00E-003 SEC;	Execute RUNTEST instruction
0x03 0xE0 0x02 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x19 0x14 0x0F	SDR 352 TDI (0000000000000 00000000000000000000000000 00000000000000000000000000 000000000000000000000000) DTDO (DATA);	Verify the frame against the data in the data buffer
0x13	N/A	Terminate the repeat algorithm
0x02 0x08 0x0E 0xE8 0x0F	SIR 8 TDI (17);	Shift in USERCODE instruction
0x03 0x20 0x0E 0xFF 0xFF 0xFF 0xFF 0x19 0x14 0x0F	SDR 32 TDI (FFFFFFFF) DTDO (DATA);	Verify the USERCODE against the USERCODE in the data buffer
0x02 0x08 0x0E 0xF4 0x0F	SIR 8 TDI (2F);	Shift in PROGRAM DONE instruction
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0x78 0x0F	SIR 8 TDI (1E);	Shift in DISABLE instruction
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x17	N/A	End VME Algorithm

See Also ▶JTAG Slim VME Embedded Source Code

▶ VME Data Format

▶ Generating JTAG Slim VME Embedded Files

VME Data Format

While the VME algorithm file contains the programming algorithm of the device, the VME data file contains the fuse and USERCODE patterns.

The first byte in the file indicates whether the data file has been compressed. A byte of **0x00** indicates that no compression was selected, and **0x01** indicates that compression was selected. When compression has been selected, each frame is preceded by a frame compression byte to indicate whether the frame is compressible. This is necessary because even though you might elect to compress the VME data file, it is possible that a compressed frame will actually be larger than an uncompressed frame. When that happens, the frame is not compressed at all and the frame compression byte of **0x00** notifies the processor that no compression was performed on the frame.

When compression has not been selected, the VME data file becomes a direct translation from the data sections of the SVF file. The END_FRAME byte, **0x10**, is appended to the end of every frame as a means for the processor to verify that the frame has indeed reached the end.

Uncompressed VME Data Format	Compressed VME Data Format
`0x00`	`0x01`
`<Frame 1>0x10`	`<Compress Byte><Frame 1>0x10`
`<Frame 2>0x10`	`<Compress Byte><Frame 2>0x10`
…	…
`<Frame N>0x10`	`<Compress Byte><Frame N>0x10`

The compression scheme used is based on the consecutive appearance of the **0xFF** byte in a frame. This byte is ubiquitous because an all **0xFF** data file is a blank pattern. When a consecutive number of ***n*** **0xFF** bytes are encountered, the VME data file will have the byte **0xFF** followed by the number ***n*** converted to hexadecimal, where ***n*** cannot exceed 255. For example, if the following were a partial data frame

```
FFFFFFFFFFFFFFFFFFFF12FFFFFF
```

the resulting compressed data would be

```
0xFF 0x0A 0x12 0xFF 0x03
```

When the processor encounters the first byte **0xFF**, it gets the next byte to determine how many times **0xFF** is compressed. The next byte is **0x0A**, which is ten in hexadecimal. This instructs the processor that **0xFF** is compressed ten times. The following byte is **0x12**, which is processed as it is. The next byte is again **0xFF** followed by **0x03**, which instructs the processor that **0xFF** is compressed three times.

See Also ▶JTAG Slim VME Embedded Source Code

▶ VME Algorithm Format

▶ Generating JTAG Slim VME Embedded Files

VME Required User Changes

To make the JTAG Full VME Embedded or JTAG Slim VME Embedded software work on your target system, you need to modify the following C functions in the hardware.c source code.

Timer The engine requires the ability to delay for fixed time periods. The minimum granularity of the delay is 1 microsecond. You can determine the type of delay. This can be a simple software timing loop, a hardware timer, or an operating system call, for example, sleep().

Port Initialization The firmware needs to place the port I/O into a known state. The programming software assumes this has occurred.

Get Data Byte The engine calls the GetByte() function to collect one byte from the VME or CPU bytestream.

Modify Port Register The engine, as it parses the bitstream data, changes an in-memory copy of the data to be written onto the I/O pins. Calls to this function do not modify the I/O pins. The engine uses *virtual types* (for example, DONE_PIN) which this function turns into physical I/O pin locations (for example, 0x400).

Output Data Byte The engine calls this function to write the in-memory copy onto the I/O pins.

Input Status This function is used by the engine to read back programming status information. The function translates physical pin locations (for example, 0x400) into *virtual types* used by the engine (for example, DONE_PIN).

Output Configuration Pins Some systems may wish to use the FPGA CFG pins. There is a separate function call to manipulate the CFG pins.

Bitstream Initialization You must determine how you plan to get the bitstream into your memory system, pre-compiled, HEX file based, or dynamically installed. Whichever method you use the data structures which pin to the bitstream need to be initialized prior to the first GetByte function call.

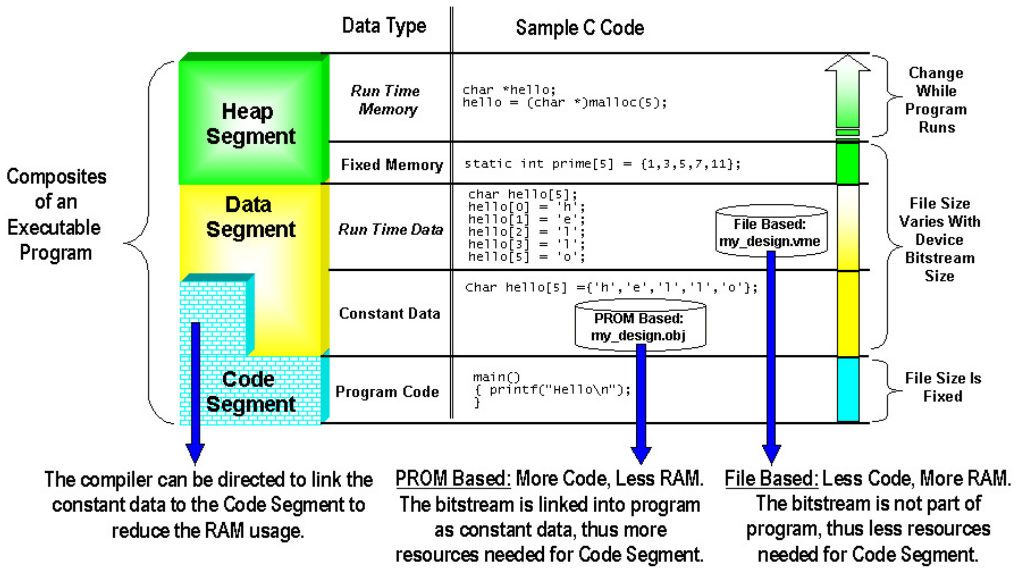
See Also ▶Customizing for the Target Platform

▶ VME Source Code

▶ JTAG Slim VME Embedded Source Code

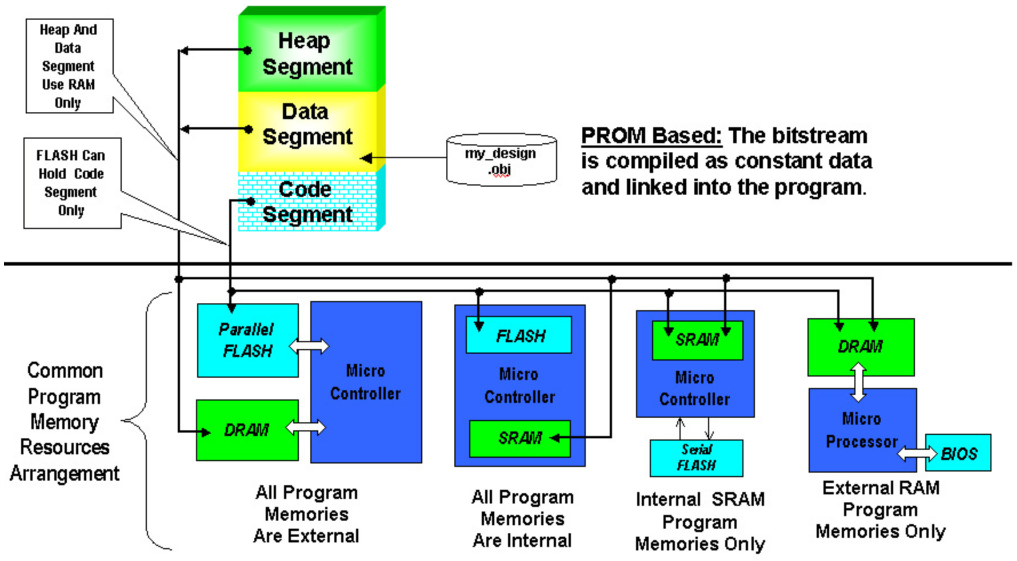
Program Memory Requirement

The following figure illustrates the JTAG Slim VME Embedded program memory requirement.


Data Type
Sample C Code
Composites of an Executable Program
Heap Segment
Data Segment
Code Segment
Run Time Memory
char *hello;
hello = (char *)malloc(5);
Fixed Memory
static int prime[5] = {1,3,5,7,11};
Run Time Data
char hello[5];
hello[0] = 'h';
hello[1] = 'e';
hello[2] = 'l';
hello[3] = 'l';
hello[5] = 'o';
File Based: my_design.vme
Constant Data
Char hello[5] ={'h','e','l','l','o'};
PROM Based: my_design.obj
Program Code
main()
{ printf("Hello\n");
}
Change While Program Runs
File Size Varies With Device Bitstream Size
File Size Is Fixed
The compiler can be directed to link the constant data to the Code Segment to reduce the RAM usage.
PROM Based: More Code, Less RAM. The bitstream is linked into program as constant data, thus more resources needed for Code Segment.
File Based: Less Code, More RAM. The bitstream is not part of program, thus less resources needed for Code Segment.

Program Memory Allocation

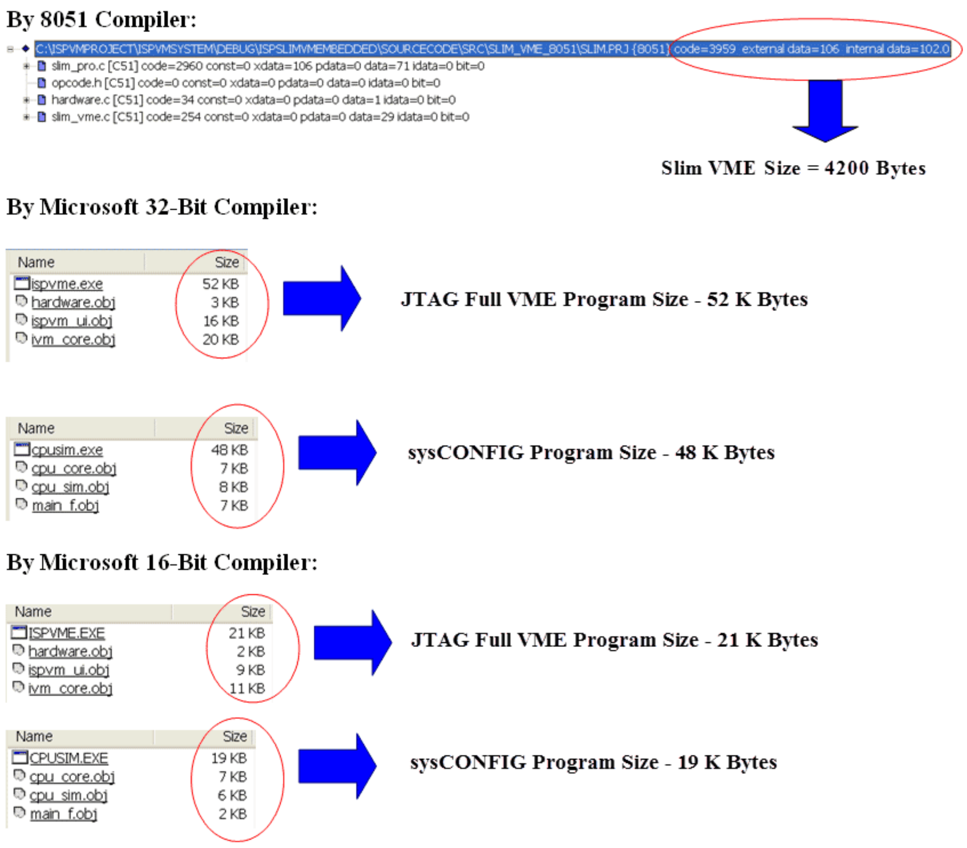
The following figure illustrates the JTAG Slim VME Embedded program memory allocation.


Heap And Data Segment Use RAM Only
FLASH Can Hold Code Segment Only
Heap Segment
Data Segment
Code Segment
my_design .obj
PROM Based: The bitstream is compiled as constant data and linked into the program.
Common Program Memory Resources Arrangement
Parallel FLASH
DRAM
Micro Controller
All Program Memories Are External
FLASH
Micro Controller
SRAM
All Program Memories Are Internal
SRAM
Micro Controller
Serial FLASH
Internal SRAM Program Memories Only
DRAM
Micro Processor
BIOS
External RAM Program Memories Only

Sample Program Size

This page provides sample program size for JTAG Full VME Embedded, JTAG Slim VME Embedded, and sysCONFIG Embedded.

Embedded Tool / Bitstream Location	JTAG Port			non-JTAG Port		Total
	JTAG Full VME Embedded		JTAG Slim VME Embedded	sysCONFIG Embedded		
	32-Bit	16-Bit	8-Bit	32-Bit	16-Bit	
File Based (Bitstream File External)	52KB	21KB	4.2KB	48KB	19KB	As Shown
PROM Based (Bitstream File Integrated)	52KB	21KB	4.2KB	48KB	19KB	As Shown + VME File Size



By 8051 Compiler:

```
C:\ISPVMPROJECT\ISPVMSYSTEM\DEBUG\ISPSLIMVMEMBEDDED\SOURCECODE\SRC\SLIM_VME_8051\SLIM.PRJ {8051} code=3959  external data=106  internal data=102.0
  slim_pro.c [C51] code=2960 const=0 xdata=106 pdata=0 data=71 idata=0 bit=0
  opcode.h [C51] code=0 const=0 xdata=0 pdata=0 data=0 idata=0 bit=0
  hardware.c [C51] code=34 const=0 xdata=0 pdata=0 data=1 idata=0 bit=0
  slim_vme.c [C51] code=254 const=0 xdata=0 pdata=0 data=29 idata=0 bit=0
```

Slim VME Size = 4200 Bytes

By Microsoft 32-Bit Compiler:

Name	Size
ispvme.exe	52 KB
hardware.obj	3 KB
ispvm_ui.obj	16 KB
ivm_core.obj	20 KB

JTAG Full VME Program Size - 52 K Bytes

Name	Size
cpusim.exe	48 KB
cpu_core.obj	7 KB
cpu_sim.obj	8 KB
main_f.obj	7 KB

sysCONFIG Program Size - 48 K Bytes

By Microsoft 16-Bit Compiler:

Name	Size
ISPVME.EXE	21 KB
hardware.obj	2 KB
ispvm_ui.obj	9 KB
ivm_core.obj	11 KB

JTAG Full VME Program Size - 21 K Bytes

Name	Size
CPUSIM.EXE	19 KB
cpu_core.obj	7 KB
cpu_sim.obj	6 KB
main_f.obj	2 KB

sysCONFIG Program Size - 19 K Bytes

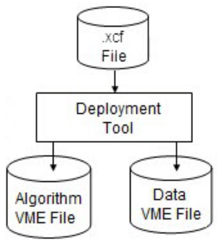
VME File Size

Refer to “Using JTAG Full VME Embedded” on page 47 for a table that compares VME file sizes taking typical Lattice devices for examples.

Generating JTAG Slim VME Embedded Files

The Slim VME files can be generated by using Deployment Tool as described as follows.

Figure 46: Slim VME File Generation Flow



In Programmer, create a project, and add the target Lattice IEEE 1532 compliant devices into the chain with the appropriate operations and data files. Refer to Programmer online help for more information on how to use Programmer.

1. Save the Programmer project (.xcf).
2. In Deployment Tool, choose **Create New Deployment.**
3. For Function Type, choose **Embedded System**.
4. For Output Type, choose **JTAG Slim VME Embedded**, then click **OK**.
5. In the Step 1 of 4 dialog box, select the XCF file, and click **Next.**
6. To have the software check for a USERCODE match between the device and the VME file before programming, select the Verify USERCODE, Program Device if Fails option.

Note

Synchronize Enable and Disable has been turned on while using Sequential mode, the software will force the VME file into Turbo mode.

7. In the Step 2 of 4 dialog box, elect the desired file options. For detailed option descriptions, including the option that allows you to generate a HEX (.c) file, see the Deployment Tool online help. To significantly reduce the ROM required for storing the VME Data buffer in the embedded system, select **Compress VME File**.
8. Click **Next**.
9. In the Step 3 of 4 dialog box, in the Output File box, specify the location and file name of the VME algorithm and data files.
10. Click **Next**.
11. In the Step 4 of 4 dialog box, click **Generate**.

Deployment Tool generates the VME files depending upon the options you have chosen, and returns a message indicating that the process succeeded or failed.

JTAG Slim VME Embedded Source Code

Both the PC and 8051-based JTAG Slim VME Embedded source code can be found in the installation path of Programmer under the <*install_path*>\embedded_source\slimembedded\sourcecode directory.

Each project requires the following files:

slim_vme.c

The file slim_vme.c is the only file to differ between the PC-based and 8051-based embedded solutions. This difference is due to the way each interfaces to the VME Algorithm and Data files through the entry point. This file contains the main and entry point functions.

slim_pro.c

The file slim_pro.c provides the programming engine of the JTAG Slim VME Embedded. The engine operates on the commands in the VME Algorithm, and fetches data from the VME Data if necessary. The engine is responsible for functions such as sending data, verifying data, observing timing delay, stepping through the state machine, decompression, and so on.

hardware.c

The only file that should be modified by the user is hardware.c. This file contains the functions to read and write to the port and the timing delay function. The user must update these functions to target the desired hardware being used. The released version targets the parallel port of the PC at address 0x0378 using Lattice's download cable.

opcode.h

The file opcode.h contains the definitions of the byte codes used in the VME Algorithm format and programming engine.

debug.h

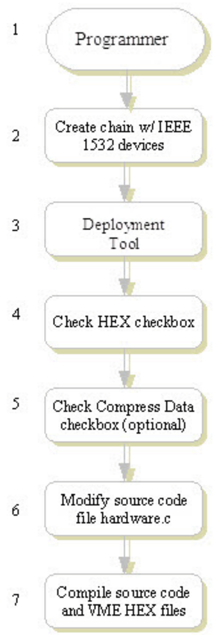
The file debug.h prints out debugging information if the preprocessor switch VME_DEBUG were defined in the project. This is an optional file to include.windriver.c and windriver.h

The files windriver.c and windriver.h target the JTAG Slim VME Embedded to Windows 95 and 98. These files are compiled if the preprocessor switch VME_WINDOWS were defined in the project file. These files should be omitted when compiling the 8051-based JTAG Slim VME Embedded onto an embedded platform.

8051 JTAG Slim VME Embedded User Flow

This appendix details the steps the user must take to use the 8051-based JTAG Slim VME Embedded.

Figure 47: 8051 JTAG Slim VME Embedded User Flow


1
Programmer
2
Create chain w/ IEEE 1532 devices
3
Deployment Tool
4
Check HEX checkbox
5
Check Compress Data checkbox (optional)
6
Modify source code file hardware.c
7
Compile source code and VME HEX files

Step 1. Create Chain with Lattice IEEE 1532 Compliant Devices using Programmer

Using Programmer, add the target IEEE 1532 compliant devices into the chain with the appropriate operations and data files. All the devices in the chain must be IEEE 1532 compliant. For more information on supporting non-Lattice devices, see Programmer’s on-line help documentation.

Step 2. Generate VME File

Use the Deployment Tool to generate the VME file. By checking the HEX check box, the VME Algorithm and Data files will be generated as C-programming files with the Algorithm and Data stored in C-style byte buffers. Refer to Deployment Tool online help for more information on using the Deployment Tool.

Step 4. Modify Source Code File hardware.c

The 8051-based source code files are written in ANSI C and can be found in the installation path of Programmer under the <*install_path*>\embedded_source\slimembedded\sourcecode\slim_vme_8051 directory. The file hardware.c is the only file that is required to be modified by the user. The user must modify the following functions according to the target platform:

- readPort
- writePort
- ispVMDelay

The following are optional functions that the user may wish to modify in order to enable and disable the hardware conditions before and after processing:

- EnableHardware
- DisableHardware

Step 5. Compile Source Code and VME HEX Files

Combine the source code and VME HEX files generated into a project to be compiled. This may be done by using a microcontroller development tool to create the project. The source codes windriver.c, windriver.h, and debug.h shall not be required to be added into the project.

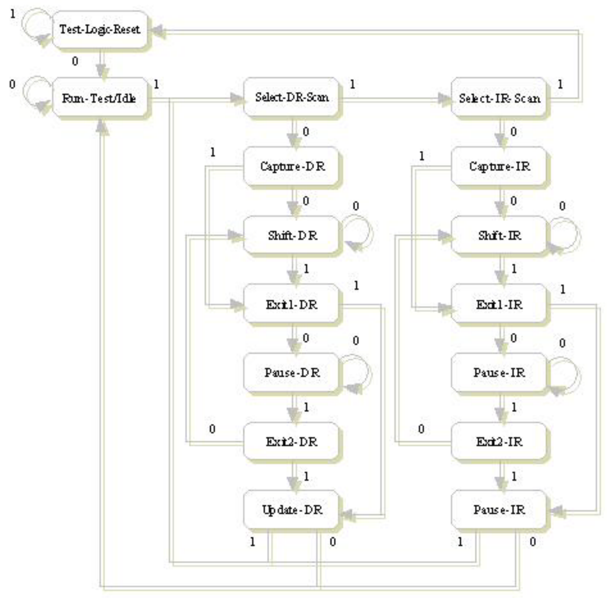
Programming Engine Flow

The programming engine of the JTAG Slim VME Embedded is driven by the byte codes of the VME Algorithm file. The Algorithm byte codes instruct the programming engine as to what sequence of functions to follow in order to shift in instructions, move the TAP controller state machine, shift data in and out of the device, and observe delays.

The TAP controller is a synchronous state machine that is based on the TMS (Test Mode Select) and TCK (Test Clock) signals of the TAP and controls the sequence of operations of the circuitry defined by the IEEE 1149.1 standard. The TCK signal can be driven at a maximum of 25 MHz for current Lattice IEEE 1532 Compliant devices.

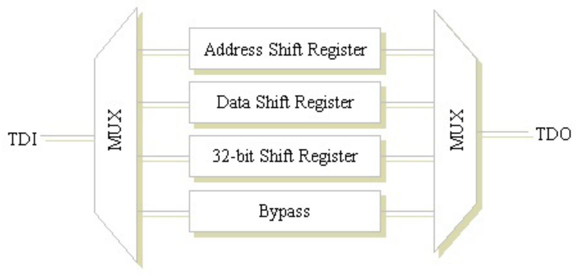
In the Shift-DR state, a decoder is present to select which shift register is enabled and connects it between TDI and TDO. The following are the shift registers: Address Shift Register, Data Shift Register, 32-bit Shift Register, and Bypass. The 32-bit Shift Register is used to store the ID Code and

Figure 48: TAP Controller State Diagram



USERCODE. The first bit clocked into any of the registers is placed into the MSB, and data is shifted one bit towards TDO as additional bits are clocked in.

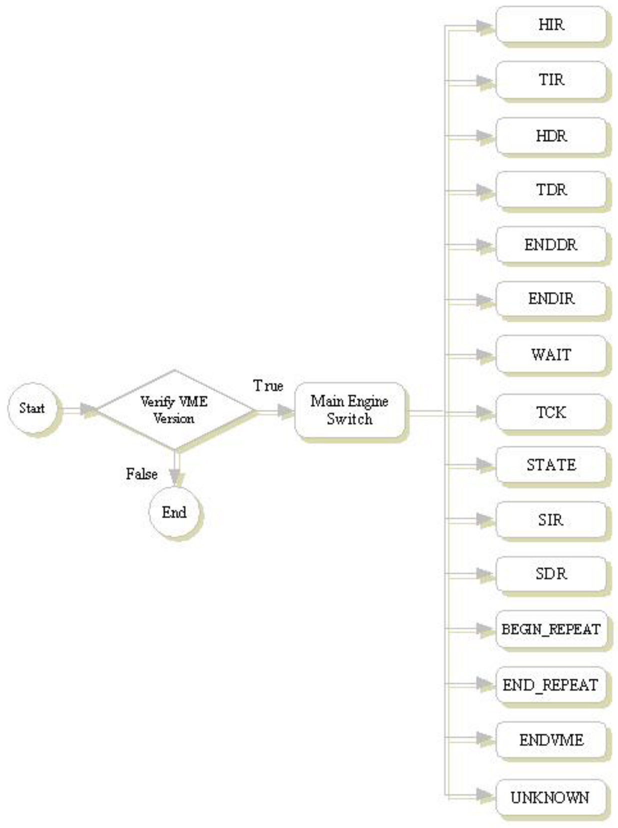
Figure 49: Shift Registers



The engine core is implemented as a switch statement. The cases in this switch statement perform specialized functions based on the byte code and its operand(s). These functions may end up calling other switch statements, calling the engine core recursively, setting global variable values, or interfacing with the device directly. Once the byte code instruction has been executed, it returns back to the main switch statement to process the next byte.

The processor begins by verifying the VME version of the algorithm file. The version is an eight byte ASCII of the format _SVME<Major Version>.<Minor Version>, where `<Major Version>` and `<Minor Version>` are digits 0-9. If the version verification fails, the processor returns the error code ERR_WRONG_VERSION, or -4.

Figure 50: Main Engine Switch

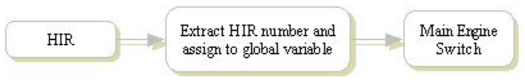


The Main Engine Switch calls the appropriate case statements based on the incoming byte code from the VME Algorithm buffer. Unrecognized byte codes will trigger the UNKNOWN case statement.

HIR Case Statement

The HIR case statement expects a number following the HIR byte code to represent the number of header devices. The number will be stored in a global variable and the processor will issue this number of bypasses prior to executing an SIR instruction.

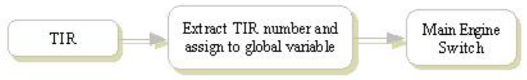
Figure 51: HIR Case Statement



TIR Case Statement

The TIR case statement expects a number following the TIR byte code to represent the number of trailer devices. The number will be stored in a global variable and the processor will issue this number of bypasses after executing an SIR instruction.

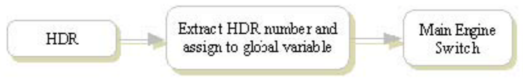
Figure 52: TIR Case Statement



HDR Case Statement

The HDR case statement expects a number following the HDR byte code to represent the number of header register bits. The number will be stored in a global variable and the processor will issue this number of bypasses prior to executing an SDR instruction.

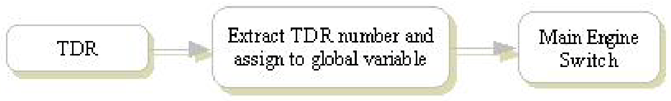
Figure 53: HDR Case Statement



TDR Case Statement

The TDR case statement expects a number following the TDR byte code to represent the number of trailer register bits. The number will be stored in a global variable and the processor will issue this number of bypasses after executing an SDR instruction.

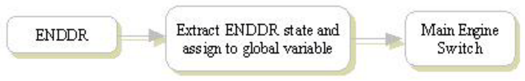
Figure 54: TDR Case Statement



ENDDR Case Statement

The ENDDR case statement expects a state following the ENDDR byte code to represent the TAP state that the processor shall move the device into after an SDR instruction. This state will be stored in a global variable.

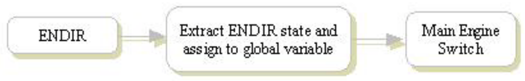
Figure 55: ENDDR Case Statement



ENDIR Case Statement

The ENDIR case statement expects a state following the ENDIR byte code to represent the TAP state that the processor shall move the device into after an SIR instruction. This state will be stored in a global variable.

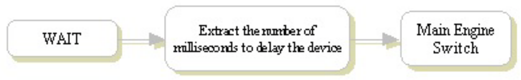
Figure 56: ENDIR Case Statement



WAIT Case Statement

The WAIT case statement expects a number following the WAIT byte code to represent the number of milliseconds of delay the device must observe. The delay is observed immediately. The user must update the delay function in the source code to make the target embedded system observe the delay duration correctly.

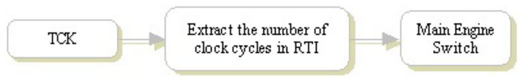
Figure 57: WAIT Case Statement



TCK Case Statement

The TCK case statement expects a number following the TCK byte code to represent the number of clocks that the device must remain in the run test idle state. The clock cycles are executed immediately into the device.

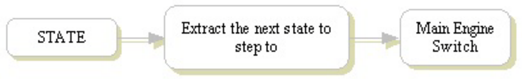
Figure 58: TCK Case Statement



STATE Case Statement

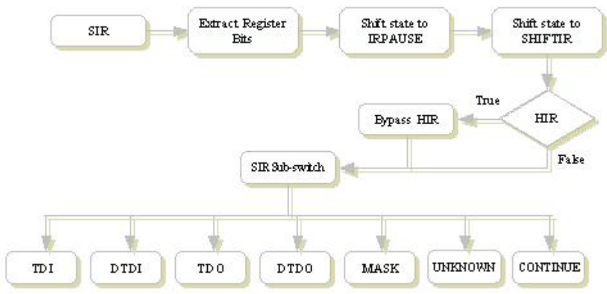
The STATE case statement expects a state following the STATE byte code to instruct the processor to step the IEEE 1149.1 bus to the next state. The state must be a valid stable state, which is IRPAUSE, DRPAUSE, RESET, or IDLE

Figure 59: STATE Case Statement



SIR Case Statement

Figure 60: SIR Case Statement



The SIR case statement begins by extracting the size of the register. The size will be used later to indicate how many bits of data will be sent or read back from the device. The processor then shifts the device into the safe state IRPAUSE, and then to the state SHIFTIR. If HIR exists (see HIR Case Statement), then the processor will bypass the HIR. The SIR sub-switch is a switch that is based off of the byte codes that can potentially be found after the SIR byte code.

The TDI byte code indicates that there is data that needs to be shifted into the device. The processor will set the TDI index variable to point to the location where the TDI data begins in the algorithm buffer.

The DTDI byte code indicates that there is data to that needs to be shifted into the device. Unlike the TDI byte code, the DTDI byte code signals that the data will be coming from the data buffer. If the data buffer has compression turned on, the first byte of the data frame will be checked to see if the frame was indeed compressible.

The TDO byte code indicates that there is data that needs to be read and verified from the device. The processor will set the TDO index variable to point to the location where the TDO data begins in the algorithm buffer.

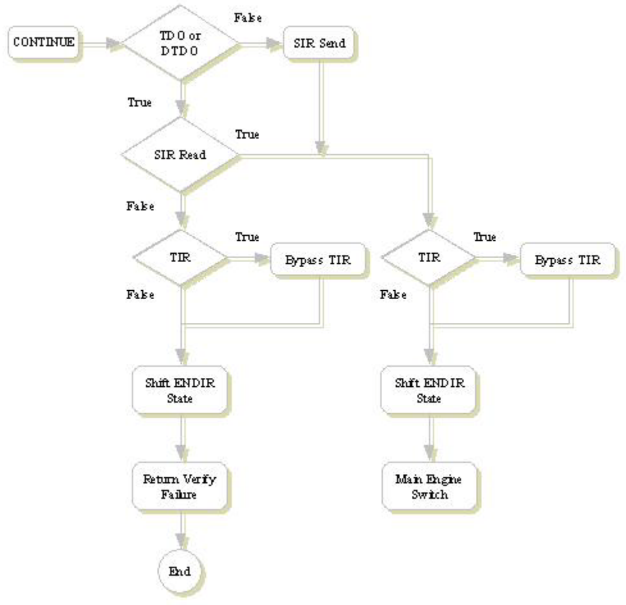
The DTDO byte code indicates that there is data that needs to be read and verified from the device. Unlike the TDO byte code, the DTDO byte code signals that the data will be coming from the data buffer. If the data buffer has compression turned on, the first byte of the data frame will be checked to see if the frame were indeed compressible.

The MASK byte code indicates that there is mask data that needs to be used when comparing the TDO values against the actual values scanned out of the device. The processor will set the MASK index variable to point to the location where the MASK data begins in the algorithm buffer.

The UNKNOWN case statement is the default for unrecognized byte codes. This case returns the error code ERR_ALGO_FILE_ERROR, or -5, to indicate an error in the algorithm.

The CONTINUE byte code terminates the SIR instruction. When this byte is encountered, it indicates that the TDI, DTDI, TDO, DTDO, and MASK indexes are pointing to their correct locations and the processor is ready to send or read and verify data from the device.

Figure 61: SIR Case Statement Continued



If the TDO or DTDO byte code were encountered in the SIR sub-switch, then that indicates that the SIR instruction is going to read data from the device, else the SIR instruction is going to send data to the device.

If reading and verifying data from the device were successful, the processor checks if TIR exists (see TIR Case Statement). If TIR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDIR byte code (see ENDIR Case Statement). The control returns back to the Main Engine Switch.

If reading and verifying data from the device were unsuccessful, the processor checks if TIR exists. If TIR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDIR byte code. The error code ERR_VERIFY_FAIL, or -1, is returned and the program exits.

If TDO or DTDO were not encountered in the SIR sub-switch, then the processor sends data to the device. If TIR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDIR byte code. The control returns back to the Main Engine Switch.

SDR Case Statement

The SDR case statement works similar to the SIR. It begins by extracting the size of the register. The size will be used later to indicate how many bits of data will be sent or read back from the device. The processor then shifts the device into the safe state DRPAUSE, and then to the state SHIFTDR. If HDR exists (see HDR Case Statement), then the processor will bypass the HDR. The SDR sub-switch is a switch that is based off of the byte codes that can potentially be found after the SDR byte code.

The TDI byte code indicates that there is data that needs to be shifted into the device. The processor will set the TDI index variable to point to the location where the TDI data begins in the algorithm buffer.

The DTDI byte code indicates that there is data to that needs to be shifted into the device. Unlike the TDI byte code, the DTDI byte code signals that the data will be coming from the data buffer. If the data buffer has compression turned on, the first byte of the data frame will be checked to see if the frame were indeed compressible.

The TDO byte code indicates that there is data that needs to be read and verified from the device. The processor will set the TDO index variable to point to the location where the TDO data begins in the algorithm buffer.

The DTDO byte code indicates that there is data that needs to be read and verified from the device. Unlike the TDO byte code, the DTDO byte code signals that the data will be coming from the data buffer. If the data buffer has compression turned on, the first byte of the data frame will be checked to see if the frame were indeed compressible.

The MASK byte code indicates that there is mask data that needs to be used when comparing the TDO values against the actual values scanned out of the device. The processor will set the MASK index variable to point to the location where the MASK data begins in the algorithm buffer.

The UNKNOWN case statement is the default for unrecognized byte codes. This case returns the error code ERR_ALGO_FILE_ERROR, or -5, to indicate an error in the algorithm.

The CONTINUE byte code terminates the SDR instruction. When this byte is encountered, it indicates that the TDI, DTDI, TDO, DTDO, and MASK indexes are pointing to their correct locations and the processor is ready to send or read and verify data from the device.

.

Figure 62: SDR Case Statement

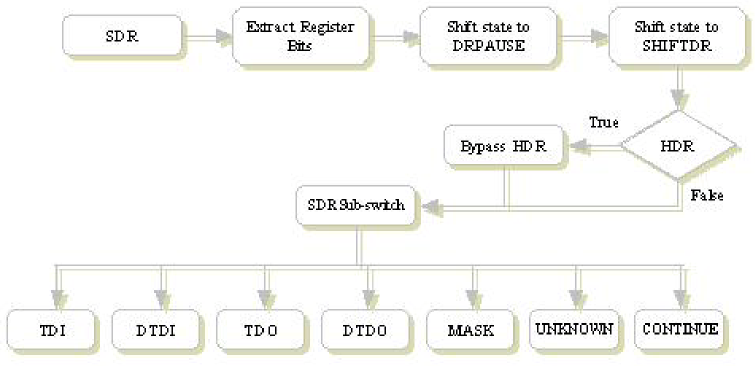
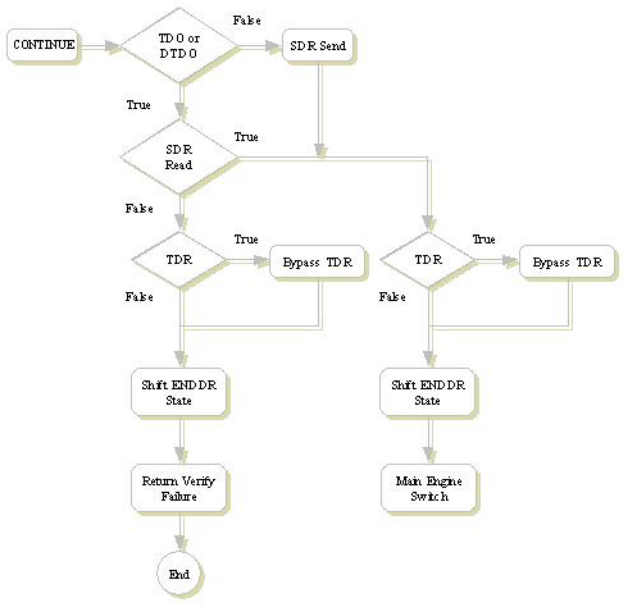


Figure 63: SDR Case Statement Continued



If the TDO or DTDO byte code were encountered in the SDR sub-switch, then that indicates that the SDR instruction is going to read data from the device, else the SDR instruction is going to send data to the device.

If reading and verifying data from the device were successful, the processor checks if TDR exists (see TDR Case Statement). If TDR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDDR byte code (see ENDDR Case Statement). The control returns back to the Main Engine Switch.

If reading and verifying data from the device were unsuccessful, the processor checks if TDR exists. If TDR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDDR byte code. The error code ERR_VERIFY_FAIL, or -1, is returned and the program exits.

If TDO or DTDO were not encountered in the SDR sub-switch, then the processor sends data to the device. If TDR exists, then the trailer devices must be bypassed. Next it shifts the device to the stable state that followed the ENDDR byte code. The control returns back to the Main Engine Switch.

BEGIN_REPEAT Case Statement

The BEGIN_REPEAT byte code makes it possible to loop the programming algorithm, thus requiring less ROM to hold the algorithm. Programming each frame requires one pass through the repeat loop. The ROM saved is substantial when one considers that a device can have several thousand frames. Instead of extrapolating the set of byte codes needed to program the frame several thousand times, only one set will be sufficient.

The BEGIN_REPEAT case statement begins by extracting the repeat size. The repeat size is typically the number of frames in the device that is to be programmed. After the repeat size has been obtained, the next byte to extract is the PROGRAM or VERIFY token. If the PROGRAM byte were present, then a pointer must be set in the data buffer to designate the beginning of the programming data. If the VERIFY byte were present, then the processor must return to the beginning location of the data buffer. This method allows programming and verification to use one set of data, thus reducing the ROM required to hold the data buffer by half.

While the repeat size, or number of un-programmed frames, is greater than zero, the algorithm index is set to point to the beginning of the repeat and a recursive call is made to the Main Engine Switch to program the frame. When the frame is processed, the Main Engine Switch returns the control to the BEGIN_REPEAT case statement. The repeat size is decremented and the process repeats until there are no frames left. The control then returns to the Main Engine Switch. While in the repeat loop, any errors such as verification or algorithm errors would result in the repeat loop returning the error code and the program would exit.

END_REPEAT Case Statement

The END_REPEAT case statement works alongside the BEGIN_REPEAT case statement. When the END_REPEAT byte code is encountered, it returns the control to the caller, which is the recursive call made by BEGIN_REPEAT. The END_REPEAT byte code appears at the end of the set of byte codes needed to program a frame.

Figure 64: BEGIN_REPEAT Case Statement

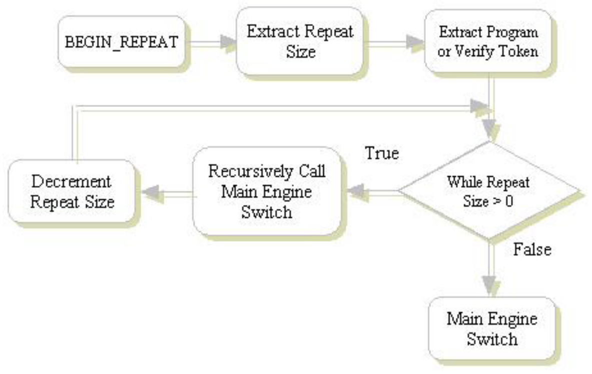
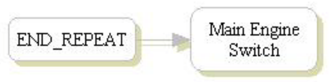


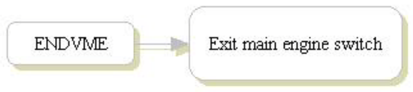
Figure 65: END_REPEAT Case Statement



ENDVME Case Statement

The ENDVME case statement is the only case where the program can return a passing value. The case statement checks if HDR exists (see HDR Case Statement). If HDR exists, then that indicates that there are still header devices that need to be programmed, thus the control returns to the Main Engine Switch. If HDR does not exist, the return value is returned to the caller, which is the entry point function and the program ends.

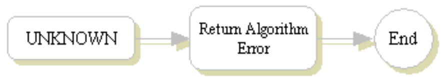
Figure 66: ENDVME Case Statement



UNKNOWN Case Statement

The UNKNOWN case statement is the default for unrecognized byte codes. This case returns the error code ERR_ALGO_FILE_ERROR, or -5, to indicate an error in the algorithm.

Figure 67: UNKNOWN Case Statement



VME Algorithm and Format

The VME Algorithm and Data files are created by deconstructing an SVF file. An SVF file is an ASCII file that contains the programming algorithm and data needed to program the device. The programming algorithm is described by statements that control the IEEE 1149.1 bus operations. When generating the VME files, Deployment Tool separates the algorithm and data into the VME Algorithm and Data files, respectively.

VME Algorithm Format

The VME Algorithm file is similar to the SVF file with the following differences:

- VME Algorithm uses byte codes from the table below to represent SVF commands
- Fuse data and USERCODE have been removed
- Looping algorithm

The following is an example of an EPV VME Algorithm file and the SVF translation for the LC4064V device:

Table 3: VME Algorithm Example

VME Algorithm Format	Serial Vector Format (SVF)	Description
0x0A 0x00	HDR 0;	
0x08 0x00	HIR 0;	
0x0B 0x00	TDR 0;	
0x09 0x00	TIR 0;	
0x06 0x03	ENDDR DRPAUSE;	
0x07 0x02	ENDIR IRPAUSE;	
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0x68 0x0F	SIR 8 TDI (16);	Shift in the IDCODE instruction
0x01 0x01	STATE IDLE;	
0x03 0x20 0x0E 0xFF 0xFF 0xFF 0xFF 0x11 0xC2 0x09 0x01 0x80 0x12 0xFF 0xFF 0xF0 0x0F	SDR 32 TDI (FFFFFFFF) TDO (01809043) MASK (0FFFFFFF);	Verify the IDCODE
0x02 0x08 0x0E 0x38 0x0F	SIR 8 TDI (1C);	Shift in the PRELOAD instruction
0x03 0x44 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x0F	SDR 68 TDI (00000000000000000);	Shift all zero data into boundary scan cells
0x02 0x08 0x0E 0xA8 0x0F	SIR 8 TDI (15);	Shift in ENABLE instruction
0x01 0x01 0x04 0x03 0x05 0x14 0x01 0x01	RUNTEST IDLE 3 TCK 2.00E-002 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0xC0 0x0F	SIR 8 TDI (03);	Shift in ERASE instruction

Table 3: VME Algorithm Example (Continued)

0x01 0x01 0x04 0x03 0x05 0x64 0x01 0x01	RUNTEST IDLE 3 TCK 1.00E-001 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0x84 0x0F	SIR 8 TDI (21);	Shift in ADDRESS INIT instruction
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0xE4 0x0F	SIR 8 TDI (27);	Shift in PROGRAM INCR instruction
0x0C 0x5F 0x15	N/A	Begin PROGRAM repeat loop of size 95
VME Algorithm Format	**Serial Vector Format (SVF)**	**Description**
0x03 0xE0 0x02 0x18 0x14 0x0F	SDR 352 DTDI (DATA);	Notice the forth byte is 0x18, which is actually DTDI. DTDI instructs the processor to send in data from the data buffer
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x13	N/A	Terminate the repeat algorithm
0x02 0x08 0x0E 0x58 0x0F	SIR 8 TDI (1A);	Shift in PROGRAM USERCODE instruction
0x03 0x20 0x18 0x14 0x0F	SDR 32 DTDI (DATA);	Shift in the USERCODE The USERCODE can be found in the data buffer.
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x02 0x08 0x0E 0x80 0x0F	SIR 8 TDI (01);	Shift in ADDRESS SHIFT instruction
0x03 0x5F 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x02 0x0F	SDR 95 TDI (400000000000000000000000);	Shift in beginning address
0x02 0x08 0x0E 0x54 0x0F	SIR 8 TDI (2A);	Shift in READ INC instruction
0x0C 0x5F 0x16	N/A	Begin VERIFY repeat loop of size 95
0x01 0x01 0x04 0x03 0x05 0x01 0x01 0x01	RUNTEST IDLE 3 TCK 1.00E-003 SEC;	Execute RUNTEST instruction
0x03 0xE0 0x02 0x0E 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x19 0x14 0x0F	SDR 352 TDI (0000000000000 00000000000000000000000000 00000000000000000000000000 00000000000000000000000) DTDO (DATA);	Verify the frame against the data in the data buffer
0x13	N/A	Terminate the repeat algorithm
0x02 0x08 0x0E 0xE8 0x0F	SIR 8 TDI (17);	Shift in USERCODE instruction
0x03 0x20 0x0E 0xFF 0xFF 0xFF 0xFF 0x19 0x14 0x0F	SDR 32 TDI (FFFFFFFF) DTDO (DATA);	Verify the USERCODE against the USERCODE in the data buffer
0x02 0x08 0x0E 0xF4 0x0F	SIR 8 TDI (2F);	Shift in PROGRAM DONE instruction
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction
0x01 0x01	STATE IDLE;	
0x02 0x08 0x0E 0x78 0x0F	SIR 8 TDI (1E);	Shift in DISABLE instruction
0x01 0x01 0x04 0x03 0x05 0x0D 0x01 0x01	RUNTEST IDLE 3 TCK 1.30E-002 SEC;	Execute RUNTEST instruction

Table 3: VME Algorithm Example (Continued)

0x17	N/A	End VME Algorithm

Customizing for the Target Platform

The source code files are written in ANSI C. The VME source codes are located in the <*install_path*>\embedded_source\vmembedded directory. The JTAG Slim VME Embedded source codes can be found in the <*install_path*>\embedded_source\slimembedded directory.

The main routines that will require customization are in the hardware.c file. It includes the routines for reading from and writing to the JTAG pins and a delay routine. These routines are well commented in hardware.c and are at the top of the file. In readPort(), a byte of data is read from the input port. In writePort(), a byte of data is written to the output port. In ispVMDelay(), the system delays for the specified number of milliseconds or microseconds. The port mapping is set at the top of the hardware.c file.

See Also ▶VME Required User Changes

Slave SPI Embedded

Slave Serial Peripheral Interface (SPI) Embedded is a high-level programming solution that enables programming FPGA families with built-in SPI port through an embedded system. This allows you to perform real-time reconfiguration to Lattice Semiconductor's FPGA families. The Slave SPI Embedded system is designed to be embedded-system independent, so it is easy to port into different embedded systems with little modifications. The Slave SPI Embedded source code is written in C code, so you may compile the code and load it to the target embedded system.

The purpose of this usage note is to provide you with information about how to port the Slave SPI Embedded source code to different embedded systems. The following sections describe the embedded system requirements and the modifications required to use Slave SPI Embedded source code.

This usage guide is updated for Slave SPI Embedded version 2.0. Major changes includes new data file format, Lattice parallel port and USB cable support.

Requirements

This section lists device requirements, embedded system requirements, and additional requirements.

Device Requirements

- Only Lattice Semiconductor's FPGA families with SPI port are supported.
- Single device support. Multiple device support is not available.
- The Slave SPI port must be enabled on the device in order to use the Slave SPI interface. This is done by setting the SLAVE_SPI_PORT to Enable using the Radiant Spreadsheet View.
- Slave SPI Configuration mode supports default setting only for CPOL and CPHA.

  CPOL - SPI Clock Polarity. Selects an inverted or non-inverted SPI clock. To transmit data between SPI modules, the SPI modules must have identical SPICR2[CPOL] values. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.

  0: Active-high clocks selected. In idle state SCK is low.

  1: Active-low clocks selected. In idle state SCK is high.

  CPHA - SPI Clock Phase. Selects the SPI clock format. In master mode, a change of this bit will abort a transmission in progress and force the SPI system into idle state.

  0: Data is captured on a leading (first) clock edge, and propagated on the opposite clock edge.

  1: Data is captured on a trailing (second) clock edge, and propagated on the opposite clock edge.

  Note

  When CPHA=1, you must explicitly place a pull-up or pull-down on SCK pad corresponding to the value of CPOL (for example, when CPHA=1 and CPOL=0 place a pull-down on SCK). When CPHA=0, the pull direction may be set arbitrarily.

Embedded System Requirements

A compiler supporting C code for the target embedded system is required.

A dedicated SPI interface that can be configured to Master SPI mode is preferred. However, if the embedded system does not have a built in SPI interface, you may consider using a general peripheral I/O ports to implement SPI functionality. In this case, minimum of four peripheral I/O's are required, with at least one of them that can be tri-stated if needed.

Read and Save operations and display operations are not supported.

Other Requirements

The Slave SPI Embedded system requires memory space to store programming data files. The storage may be internal or external memory

(RAM, Flash, etc.). You may also consider storing the programming data in an external system such as PC. In this case, you need to establish communication between the external system and the embedded system.

Slave SPI Embedded Algorithm Format

The Slave SPI algorithm file contains byte codes that represent the programming algorithm of the device or chain.

Table 4: Slave SPI Algorithm Format

SSPI Symbol	Hex Value
STARTTRAN	0x10
CSTOGGLE	0x11
TRANSOUT	0x12
TRANSIN	0x13
RUNCLOCK	0x14
TRSTTOGGLE	0x15
ENDTRAN	0x1F
MASK	0x21
ALGODATA	0x22
PROGDATA	0x25
RESETDATA	0x26
PROGDATAEH	0x27
REPEAT	0x41
ENDREPEAT	0x42
LOOP	0x43
ENDLOOP	0x44
STARTOFALGO	0x60
ENDOFALGO	0x61
HCOMMENT	0xA0
HENDCOMMENT	0xA1
ALGOID	0xA2
VERSION	0xA3
BUFFERREQ	0xA4
STACKREQ	0xA5
MASKBUFREQ	0xA6

Table 4: Slave SPI Algorithm Format (Continued)

SSPI Symbol	Hex Value
HCHANNEL	0xA7
HEADERCRC	0xA8
COMPRESSION	0xA9
HDATASET_NUM	0xAA
HTOC	0xAB

Slave SPI Embedded Data Format

While the SSPI algorithm file contains the programming algorithm of the device, the SSPI data file contains the fuse and USERCODE patterns. The first byte in the file indicates whether the data file has been compressed. A byte of **0x00** indicates that no compression was selected, while **0x01** indicates that compression was selected.

When compression has been selected, each frame is preceded by a frame compression byte to indicate whether the frame is compressible. This is necessary because even though you might elect to compress the SSPI data file, it is possible that a compressed frame will actually be larger than an uncompressed frame. When that happens, the frame is not compressed at all and the frame compression byte of **0x00** is added to notify the processor that no compression was performed on the frame.

Uncompressed Slave SPI Data Format	Compressed Slave SPI Data Format
0x00	0x01
<Frame 1>0x10	<Compress Byte><Frame 1>0x10
<Frame 2>0x10	<Compress Byte><Frame 2>0x10
…	…
<Frame N>0x10	<Compress Byte><Frame N>0x10

Generating Slave SPI Embedded Files

The Slave SPI Embedded files can be generated through Radiant Programmer. Choose **View > Embedded Options**. The Slave SPI Embedded generation dialog allows you to generate the file in hex (C compatible) array or binary. The binary Slave SPI file can be used by the PC version of Slave SPI Embedded and utilizes the extension *.sea for algorithm files, and *.sed for data files. Also, the binary file can be uploaded to internal or external memory of the embedded system if you plan to implement the system in that manner.

The hex file is a C programming language file that must be compiled with the EPROM-based version of Slave SPI Embedded processor and utilizes the extension *.c. The binary file is generated by default. Other options are available through the dialog, such as data file compression, adding comments to the algorithm file, or disable generating the algorithm or data file.

Modifications

The Slave SPI Embedded source code is installed in the <*install_path*>\embedded_source\sspiembedded\sourcecode directory where you installed the Radiant Programmer. There are two directories in the src directory: SSPIEm and SSPIEm_eprom.

The first directory, SSPIEm, contains the file-based Slave SPI Embedded source code, and can support sending and receiving multiple bytes over the channel. The second directory, SSPIEm_eprom, contains the EPROM-based Slave SPI Embedded source code, which supports the algorithm and data being compiled with the process system.

In the files that require user modification, comments surrounded by asterisks (*) will require your attention. These comments indicate that the following section may require user modification. For example:

```
//*********************************************************************
//* Example comment
//*********************************************************************
```

Before using the Slave SPI Embedded system, there are three sets of files (.c / .h) that need to be modified. The first set, hardware.c and hardware.h, must be modified. This file contains the SPI initialization, wait function, and SPI controlling functions. If you want to enable debugging functionalities, debugging utilities need to be modified in this file as well. The hardware.c source code supports transmitting and receiving multiple bytes at once. This approach may be faster in some SPI architecture, but it requires a buffer to store the entire frame data.

The second set, intrface.c and intrface.h, contains functions that utilize the data and algorithm files. There are two sections in this file that requires attention. The first one is the data section. When the processor in Slave SPI Embedded system needs to process a byte of data, it calls function `dataGetByte()`. Slave SPI Embedded version 2.0 requires data file no matter what operation it is going to process. You are responsible to modify the function to fit their configuration. The second section is the algorithm section.

In Programmer, there is an option to generate both the algorithm file and the data file in hex array format (C compatible). If you wish to compile the algorithm and data along with the Slave SPI Embedded system, use the source code in the <*install_path*>\embedded_source\sspiembedded\sourcecode\sspiem_eprom directory. Users only need to put the generated .c file in the same folder as Slave SPI Embedded system code and then compile the whole thing. If the embedded system has internal memory that can be reached by address, using EPROM version of intrface.c is also ideal. For users who plan to put the algorithm and data in external storage, intrface.c and intrface.h may need modification.

The third file set is SSPIEm.c and SSPIEm.h. Function `SSPIEm_preset()` allows you to set which algorithm and data will be processed. This function needs to be modified according to your configuration.

Below is information about functions you are responsible to modify.

hardware.c

You are responsible to modify `TRANS_transmitBytes()` and `TRANS_receiveBytes()`. If you wish to implement Slave SPI Embedded so it transmit one byte at a time, then `TRANS_tranceive_stream()` needs to be modified.

int SPI_init(); This function will be called at the beginning of the Slave SPI Embedded system. Duties may include, but not be limited to:

- Turning on SPI port
- Enabling counter for wait function
- Configuring SPI peripheral IO ports (PIO)
- Resetting SPI
- Initializing SPI mode (Master mode, channel used, etc)
- Enabling SPI

The function returns a 1 to indicate initialization successful, or a 0 to indicate fail.

int SPI_final(); This function is called at the very end of the Slave SPI Embedded system and returns a 1 to indicate success, or a 0 to indicate fail.

void wait(int ms); This function takes a delay time (in milliseconds) and waits for the amount of delay time. This function does not need a return value.

int TRANS_starttranx(unsigned char channel); This function starts an SPI transmission. Duties may include, but are not limited to:

- Pulling Chip Select (CS) low
- Starting Clock
- Flushing read buffer

If the dedicated SPI interface in the embedded system automatically starts the clock and pulls CS low, then this function only returns a 1. This function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_endtranx(); This function finalizes an SPI transmission. Duties may include, but are not limited to:

- Pulling CS high
- Terminating Clock

If the dedicated SPI interface in the embedded system automatically terminates the clock and pulls CS high, then this function only returns a 1. This function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_cstoggle(unsigned char channel); This function toggles the CS of current channel, and is called between `TRANS_starttranx()` and `TRANS_endtranx()`. It first pulls CS low, waits for a short period of time, and pulls CS high. A simple way to accomplish this is to transmit some dummy data to the device. One bit is enough to accomplish this. All one (1) for dummy is recommended, because usually the channel is held high during rest, and Lattice devices ignore opcode 0xFF (no operation). The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_trsttoggle(unsigned char toggle); This function toggles the CRESET (TRST) signal. The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_runClk(int clks); This function runs a number of extra clocks on an SPI channel. If the dedicated SPI interface does not allow free control of clock, a workaround is to enable the CS of another channel that is not being used. This way the device still sees the clock but the CS of current channel will stay high. The function returns a 1 to indicate success, or a 0 to indicate fail.

int TRANS_transmitBytes (unsigned char *trBuffer, int trCount); This function is available if you wish to implement transmitting multiple bits one byte at a time. It is responsible to transmit the number of bits, indicated by `trCount,` over the SPI port. The data to be transmitted is stored in `trBuffer.` Integer `trCount` indicates the number of bits being transmitted, which should be divisible by eight (8) to make it byte-bounded. If `trCount` is not divisible by eight, pad the least significant bits of the transmitted data with ones (1).

int TRANS_receiveBytes (unsigned char *rcBuffer, int rcCount); This function is available if you wish to implement receiving multiple bits one byte at a time. It is responsible to receive the number of bits, indicated by `rcCount`, over the SPI port. The data received may be stored in rcBuffer. Integer rcCount indicates the number of bits being received, which should be divisible by eight (8) to make it byte-bounded. If `rcCount` is not divisible by eight, pad the most significant bits of the received data with ones (1).

int TRANS_transceive_stream(int trCount, unsigned char *trBuffer, trCount2, int flag, unsigned char *trBuffer2); This function is available for modification if you wish to implement transmission with one byte at a time. The function also appears in implementation of transmission with multiple bytes at once, but you don't need to modify it.

For single byte transmission, this is the most complex function that needs to be modified. First, it will transmit the amount of bits specified in `trCount` with data stored in `trBuffer.` Next, it will have the following operations depending on the flag:

- NO_DATA: End of transmission. `trCount2` and `trBuffer2` are discarded.

- BUFFER_TX: Transmit data from `trBuffer2`.
- BUFFER_RX: Receive data and compare it with `trBuffer2`.
- DATA_TX: Transmit data from external data.
- DATA_RX: Receive data from trbuffer2.

If the data is not byte-bounded and your SPI port only transmits and receives byte-bounded data, padding bits are required to make it byte-bounded. When transmitting non-byte-bounded data, add the padding bits at the beginning of the data. When receiving data, do not compare the padding, which are at the end of the transfer. The function returns a 1 to indicate success, or a 0 to indicate fail.

(optional) int dbgu_init(); This function initializes the debugging functionality. It is up to you to implement it, and implementations may vary.

(optional) void dbgu_putint(int debugCode, int debugCode2); This function puts a string and an integer to the debugging channel. It is up to you to take advantage of these inputs.

SSPIEm.c

int SSPIEm_preset(); This function calls dataPreset() and algoPreset() functions to pre-set the data and algorithm. The input to this function depends on the configuration of the embedded system. This function returns 1 to indicate success, or 0 to indicate fail.

intrface.c - Data Section

Global Variables Global variables may vary due to different implementations.

int dataPreset(); This function allows you to set which data will be used for processing. It returns 1 to indicate success, or 0 to indicate fail.

int dataInit (unsigned char *comp); This function initializes the data. The first byte of the data indicates if the fuse data is compressed. It retrieves the first byte and stores it in the location pointed to by `*comp`. The fuse data starts at the second byte. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, the following is an example implementation:

Points variable data to the beginning of the fuse data.

Resets `count` to 0.

int dataGetByte(int *byteOut); This function gets one byte from data array and stores it in the location pointed to by byteOut. The implementation may vary due to different configuration. The function returns 1 to indicate success, or 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, here is a sample implementation:

Gets byte that variable data points to.

Points data to the next byte.

`Count++.`

int dataReset(); This function resets the data pointer to the beginning of the fuse data. Note that the first byte of the data is not part of the fuse data. It indicates if the data is compressed. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail. For implementation that uses internal memory, which can be accessed by addressing, the following is an example implementation:

Points variable data to the beginning of the data array.

Resets count to 0.

Note: This section has data utilized functions. Modification of this section is optional if you wish to compile the algorithm along with Slave SPI Embedded system.

int dataFinal(); This function is responsible to finish up the data. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

intrface.c - Algorithm Section

Global variables Global variables may vary due to different implementation.

int algoPreset(); This function allows you to set which algorithm will be used for processing. It returns 1 to indicate success, or 0 to indicate fail.

int algoInit(); This function initializes the data. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

In our implementation, it only sets `algoIndex` to 0.

int algoGetByte(unsigned char *byteOut); This function gets one byte from the algorithm bitstream and stores it in the location pointed to by `byteOut`. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

int algoFinal(); This function is responsible to finish up the algorithm. The implementation may vary due to different configuration. The function returns a 1 to indicate success, or a 0 to indicate fail.

intrface.c - Sample Configurations

There are several different options to configure the Slave SPI Embedded data file and algorithm file. The following are two possible configurations.

1. EPROM Approach

With this version, both algorithm and data are generated into C-compatible Hex array and compiled along with Slave SPI Embedded source code. Here are how the functions are modified to fit this configuration:

- Include both Hex arrays in the global scale.
- Pass the pointer to the arrays to `SSPIEm_preset()`. In this function, pass the pointer to `algoPreset()` and `dataPreset()` functions, respectively. Both functions store the pointer in global variables defined in intrface.c.
- In `algoInit()` and `dataInit()` functions, set the counters to zero (0).
- In `algoGetByte()` and `dataGetByte()` functions, read a byte from the respective array, and increment the counter.
- In `dataReset()` function, reset the counter to zero (0).
- In `algoFinal()` and `dataFinal()` functions, set the pointer to both array to NULL. This is optional.

Although optional, it is a good idea to keep track of the size of both data and algorithm arrays. The size of array may be passed to Slave SPI Embedded through the preset functions.

If the embedded system uses internal memory that can be reached the same way as using array, this configuration may also fit into the embedded system.

If you plan to use EPROM approach, intrface.c will be available, and you may not need to modify it. The files intrface.c, intrface.h, SSPIEm.c, and SSPIEm.h are in the <*install_path*>/SSPIEmbedded/SourceCode/src/SSPIEm_eprom directory.

2. File System Approach

This approach is used when implementing Slave SPI Embedded command-line executable on PC. If the embedded system has similar file system, it may access the algorithm and data through the file system. Here is how the functions are modified to fit this configuration:

- Pass the name of the algorithm and data file to `SSPIEm_preset()`. In this function, pass them to `algoPreset()` and `dataPreset()` functions, respectively. Both functions store the name of the file in global variables defined in intrface.c.
- In `algoInit()` and `dataInit()` functions, open the file and check if they are readable. If the file is not opened as a stream, set the counter to zero (0).
- In `algoGetByte()` and `dataGetByte()` functions, read a byte from the respective file, and increment the counter if needed.

- In `dataReset()` function, reset the counter to zero (0), if needed. If the file is read as a stream, rewind the stream.
- In algoFinal() and `dataFinal()` functions, close both files.

Usage

In order to use the Slave SPI Embedded system, include it in the target embedded system by adding SSPIem.h to the header list. To start the processor, simply make this function call:

```
SSPIEm(unsigned int algoID);
```

Currently, the converter does not have `algoID` capability. This capability is reserved for future use. Use 0xFFFFFFFF for `algoID`.

Return Codes from Slave SPI Embedded

The utility returns a 2 when the process succeeds, and returns a number less than or equal to 0 when it fails. Table 5 lists return codes from Slave SPI Embedded.

Table 5: Return codes from Slave SPI Embedded

Results	Return Code
Succeed	2
Process Failed	0
Initialize Algorithm Failed	-1
Initialize Data Failed	-2
Version Not Supported	-3
Header Checksum Mismatch	-4
Initialize SPI Port Failed	-5
Initialization Failed	-6
Algorithm Error	-11
Data Error	-12
Hardware Error	-13
Verification Error	-20

Programming Considerations for SSPIEM Modification with Aardvark SPI APIs

Aardvark is an SPI adapter that can be used for programming of Lattice FPGA devices with Slave SPI. The Radiant software provides SSPIEM example source codes which are modified with Aardvark SSPI APIs. However Lattice does not guarantee that these APIs will be supported for all the programming modes incorporated in the .sea files generated by the Lattice Deployment Tool, which are used by our SSPIEM source codes. This is due to the limitation of the Aardvark adapter and with its associated read/write APIs meant for the data transfer between the Lattice's algo interpretation logic and the actual programming hardware driver logic.

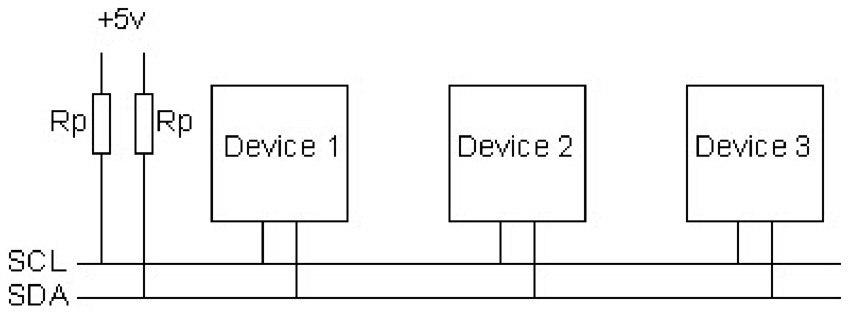
The Aardvark adapter has a buffer limitation of 4 KB and any algo file data above 4 KB will overflow the buffer and result in a programming failure.

I^{2}C Embedded

The physical I^{2}C buss consists of two wires: SCL and SDA.

- SCL is the clock line. It is used to synchronize all data transfers over the I^{2}C bus.
- SDA is the data line.

The SCL & SDA lines are connected to all devices on the I^{2}C bus. There must be a third wire connected to ground or 0 volts. There may also be a 5V wire for power distribution t he devices. Both SCL and SDA lines are "open drain" drivers, meaning that the device can drive its output low, but it cannot drive it high. For the line to be able to go high, you must provide pull-up resistors to the 5V supply. There should be a resistor from the SCL line to the 5V line and another from the SDA line to the 5V line. You only need one set of pull-up resistors for the entire I^{2}C bus, as illustrated below.

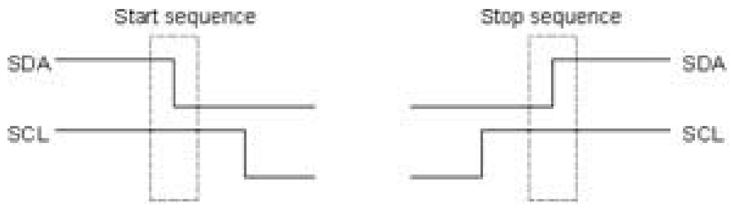


Masters and Slaves

The devices on the I^{2}C bus are either masters or slaves. The master is always the device that drives the SCL clock line. The slaves are the devices that respond to the master. Only a master can initiate a transfer over the I^{2}C bus. A slave cannot initiate a transfer over the I^{2}C bus. There can be, and usually are, multiple slaves on the I^{2}C bus. However, there is normally only one master. It is possible to have multiple masters, but it is typical and not covered in this document. For the purposes of this document, the CrossLink-NX device is always the slave.

CrossLink-NX Slave I^{2}C Programming

When the master communicates to a slave (CrossLink-NX for example) it begins by issuing a start sequence on the I^{2}C bus. A start sequence is one of two special sequences defined for the I^{2}C bus, the other being the stop sequence. The start sequence and stop sequence are the only time when the SDA (data line) is allowed to change while the SCL (clock line) is high. When data is being transferred, SDA must remain stable and not change while the SCL is high. The start and stop sequences mark the beginning and end of a transaction with the slave device.



The primary I^{2}C port of the CrossLink-NX device can be used as a user I^{2}C port function or as a device programming port. When used for device programming, the primary I^{2}C port is a slave I^{2}C with a default slave address of 7'b1000000 for 7-bit addressing or 10'b1111000000 for 10-bit addressing. The primary I^{2}C port must be enabled in order to support the device programming using the I^{2}C protocol. This is done by setting the I2C_Port preference to ENABLE in the software. The I^{2}C programming supports single device programming.

The sequence for device programming using the I^{2}C follows the standard Lattice device programming algorithm. The I^{2}C bus hardware requirements, the timing requirements, and the receive/transmit protocols must follow the standard I^{2}C specification. The definition of the SDA data time is the delay form the SCL falling edge 30% VDD to SDA falling edge 70% VDD. The SDA data setup time is the time requirement from the SDA falling edge 30% VDD to the SCL rising edge 30% VDD.

All the I^{2}C slave commands consist of one byte op-code followed by three one-byte operand, except the ISC DISABLE command. The ISC DISABLE command for I^{2}C programming consists of one byte op-code followed by two operands. I^{2}C programming can also be done in the background. In this case, the ISC ENABLE command (0XC6) should be replaced by the LSC_ENABLE_X command (0X74).

Using the PC-based I^{2}C Embedded Programming

The I^{2}C Embedded system for CrossLink-NX is a quick and easy way to validate I^{2}C files and the I^{2}C Embedded programming engine by successfully processing the target CrossLink-NX device using the FTDI USB2 Cable of the PC.

The programming algorithm of the device is separated into I^{2}C algorithm file and I^{2}C data file. This separation of the algorithm and data allows the optimization of the I^{2}C embedded programming engine. It also allows you to mix I^{2}C data files with I^{2}C algorithm files.

To access the PC-based I^{2}C Embedded System, use the Radiant Deployment Tool to add the CrossLink-NX device. Then, select the I^{2}C embedded programming options from the Generate I^{2}C dialog box. For more information, refer to the Deployment Tool online help.

The only source code file that must be modified is the hardware.c file. The source files can be found in <*install_path*>\embedded_source\i2cembedded\src\i2cem Radiant directory.

hardware.c

The only file that you should modify is hardware.c. This file contains the functions to read and write to the port and the timing delay function. You must update these functions to target the desired hardware.

opcode.h

The opcode.h file contains the definitions of the byte codes used in the I^{2}C algorithm format and programming engine.

i2c_core.c

The i2c_core.c file provides the programming engine for the I^{2}C embedded system. The engine operates on the commands in the I^{2}C algorithm, and obtains data from the I^{2}C data, if necessary. The engine is responsible for functions such as sending data, verifying data, observing timing delay, decompression, and so on.

i2c_main.c

The i2c_main.c file is the only file that differs between the PC-based and the 8051-based embedded solutions. This difference is due to the way each of these interfaces to the I2C algorithm and data files through the entry point. This file contains the main and entry point functions.

Using the 8051-based I^{2}C Programming

To program embedded systems using the 8051-based I2C programming, you must generate the I^{2}C files as HEX to create the I^{2}C algorithm and data files as C programming files. Each file contains a C programming style byte buffer that holds the I^{2}C algorithm or data.

The HEX files must be compiled along with the 8051-based I^{2}C System source code. The source code contains handles that allow the compiler to link the buffers of the hexadecimal files to the main source code. The only source code file that you need to modify is the hardware.c file. The source files can be found in the <*install_path*>\embedded_source\i2cembedded\src\i2cem_eprom directory.

hardware.c

The only file that you should modify is hardware.c. This file contains the functions to read and write to the port and the timing delay function. You must update these functions to target the desired hardware.

opcode.h

The opcode.h file contains the definitions of the byte codes used in the I^{2}C algorithm format and programming engine.

i2c_core_eprom.c

The i2c_core.c file provides the programming engine for the I^{2}C embedded system. The engine operates on the commands in the I^{2}C algorithm, and fetches data from the I^{2}C data, if necessary. The engine is responsible for functions such as sending data, verifying data, observing timing delay, decompression, and so on.

i2c_eprom.c

The i2c_main.c contains the main and entry point functions for 8051-based I^{2}C Programming.

I^{2}C Algorithm Format

The I^{2}C algorithm file contains byte codes that represent the programming algorithm of the device or chain.

Table 6: I^{2}C Algorithm Byte Codes

I2C Symbol	Hex Value
I2C_STARTTRAN	0x10
I2C_RESTARTTRAN	0x11
I2C_ENDTRAN	0x12
I2C_TRANSOUT	0x13
I2C_TRANSIN	0x14
I2C_RUNCLOCK	0x15
I2C_WAIT	0x16
I2C_LOOP	0x17
I2C_ENDLOOP	0x18
I2C_TDI	0x19
I2C_CONTINUE	0x1A
I2C_TDO	0x1B
I2C_MASK	0x1C
I2C_BEGIN_REPEAT	0x1D
I2C_END_REPEAT	0x1E
I2C_END_FRAME	0x1F
I2C_DATA	0x20
I2C_PROGRAM	0x21
I2C_VERIFY	0x22
I2C_DTDI	0x23
I2C_DTDO	0x24
I2C_COMMENT	0x25
I2C_ENDVME	0x7F

I^{2}C Data Format

While the I^{2}C algorithm file contains the programming algorithm of the device, the I^{2}C data file contains the fuse and USERCODE patterns.

The first byte in the file indicates whether the data file has been compressed. A byte of **0x00** indicates that no compression was selected, and **0x01** indicates that compression was selected. When compression has been selected, each frame is preceded by a frame compression byte to indicate whether the frame is compressible. This is necessary because even though you might elect to compress the I^{2}C data file, it is possible that a compressed frame will actually be larger than an uncompressed frame. When that happens, the frame is not compressed at all and the frame compression byte of **0x00** notifies the processor that no compression was performed on the frame.

When compression has not been selected, the I^{2}C data file becomes a direct translation from the data sections of the SVF file. The END_FRAME byte, **0x1F**, is appended to the end of every frame as a means for the processor to verify that the frame has indeed reached the end.

Uncompressed I2C Data Format	Compressed I2C Data Format
0x00	0x01
<Frame 1>0x10	<Compress Byte><Frame 1>0x10
<Frame 2>0x10	<Compress Byte><Frame 2>0x10
…	…
<Frame N>0x10	<Compress Byte><Frame N>0x10

The compression scheme used is based on the consecutive appearance of the **0xFF** byte in a frame. This byte is ubiquitous because an all **0xFF** data file is a blank pattern. When a consecutive number of **n 0xFF** bytes are encountered, the I^{2}C data file will have the byte **0xFF** followed by the number n converted to hexadecimal, where **n** cannot exceed 255. For example, if the following were a partial data frame.

FFFFFFFFFFFFFFFFFFFF12FFFFFF the resulting compressed data would be:

0xFF 0x0A 0x12 0xFF 0x03

When the processor encounters the first byte **0xFF**, it gets the next byte to determine how many times **0xFF** is compressed. The next byte is **0x0A**, which is ten in hexadecimal. This instructs the processor that **0xFF** is compressed ten times. The following byte is **0x12**, which is processed as it is. The next byte is again **0xFF** followed by **0x03**, which instructs the processor that **0xFF** is compressed three times.

I^{2}C Embedded Programming Required User Changes

To make the I^{2}C Embedded Programming software work on your target system, you need to modify the following C functions in the hardware.c source code.

Timer(SetI2Cdelay())

The engine requires the ability to delay for fixed time periods. The minimum granularity of the delay is 1 microsecond. You can determine the type of delay. This can be a simple software timing loop, a hardware timer, or an operating system call, for example, sleep().

Port Initialization

The firmware needs to place the port I/O into a known state.

SetI2CStartCondition()

This function is used to issue a start sequence on the I^{2}C Bus.

SetI2CreStartCondition()

This function is used to issue a start sequence on the I^{2}C Bus.

SetI2CStopCondition()

This function is used to issue a stop sequence on the I^{2}C Bus.

ReadBytesAndSendNACK()

This function is used to read the SDA pin from the port.

SendBytesAndCheckACK()

To apply the specified value to the SDA pin indicated.

Generating I^{2}C Files

This section describes how to generate I^{2}C files. An .xcf file is required for the CrossLink-NX FPGA.

To generate an .xcf file for the CrossLink-NX, if the .xcf file does not exist or has not yet been created:

1. Start the Radiant Programmer software and create a new Blank Project.
2. Select CrossLink-NX as Device Family.
3. Select the Device Type according to your device.
4. Choose **Edit > Device Properties,** or right click on the device, and in the dropdown menu, choose **Device Properties**.
5. In the Device Properties dialog box:

- In the Access Mode dropdown menu, choose **I^{2}C Interface Programming**.
- In the Operation dropdown menu, choose the desired operation.
- In the Programming File box, browse to your design's .jed programming file.
- In the I2C Slave Address box, enter the correct I^{2}C slave address. The default address is 0x40.

6. Chose **File > Save** or **File > Save (filename).xcf As...** and give the file a name. Ensure that the extension of the file is xcf.

To generate I^{2}C Files:

1. Start the Deployment Tool.
2. In the Getting Started dialog box, select **Create New Deployment**.
3. In the Function Type dropdown menu, choose **Embedded System**.
4. In the Output File Type dropdown menu, choose as **I^{2}C Embedded**.
5. Click **OK.**
6. In the Step 1 of 4 dialog box, browse to the .xcf file you created with the Programmer software, and select **Input XCF file(s)**.
7. Click **Next**.
8. In the Step 2 of 4 dialog box, select **Compress Embedded Files** depending upon the requirement, select **Generate Hex(.c) Files** for 8051 micro-processor usages, and click **Next**.
9. In the Step 3 of 4 dialog box, select the Algorithm File and Data File to rename and change the location of the file name. Make sure the file name has the extension .iea and .ied, respectively, and click **Next**.
10. In the Step 4 of 4 dialog box, click **Generate** to generate the files.
11. The files will be generated as shown as below.
12. The Deployment Tool project can now be saved by selecting **File > Save As**. The saved file will generate the same data file and algorithm file when loaded again.
13. Modify the Source Code File (hardware.c). The 8051-based source code files are written in ANSI C. The file hardware.c is the only file that is required to be modified by the user. The user must modify the following functions according to the target platform:

```
SetI2Cdelay()
SetI2CStartCondition()
SetI2CreStartCondition()
SetI2CStopCondition()
ReadBytesAndSendNACK( int length, unsigned char *a_ByteRead
, int NAck)
```

Where

`int length` = Number of bits to read

`*a_ByteRead` = Buffer to store byte

`int NAck` - Option to send

0 = No

1 - Yes

```
int SendBytesAndCheckACK(int length, unsigned char
*a_bByteSend
```

Where

`int length` = Number of bits to send

`*a_bByteSend` = Buffer storing data to send

The following are optional functions that the user may wish to modify in order to enable and disable the hardware conditions before and after processing:

```
EnableHardware()
DisableHardware()
```

14. Compile Source Code and I2C HEX Files. Combine the source code and I^2C HEX files generated by Deployment Tool into a project to be compiled. This may be done using a microcontroller development tool to create the project.

Modify the Delay Function

When porting Embedded I^2C to a native CPU environment, the speed of the CPU or the system clock that drives the CPU is usually known. The speed or the time it takes for the native CPU to execute one loop then can be calculated.

The for loop usually is compiled into the ASSEMBLY code as shown below:

```
LOOP: EDC RA;
       JNZ LOOP;
```

If each line of assembly code needs four (4) machine cycles to execute, the total number of machine cycles to execute the loop is 2 x 4 = 8.

Usually: system clock = machine clock (the internal CPU clock).

Note

Some CPUs have a clock multiplier to double the system clock for the machine clock.

Let the machine clock frequency of the CPU be F (in MHz), then one machine cycle = 1/F.

The time it takes to execute one loop = (1/F) x 8.

It is obvious that the formula can be transposed into one microsecond = F/8.

Example: The CPU internal clock is set to 48 MHz, then one microsecond = 48/8 = 6.

The C code shown below can be used to create the millisecond accuracy. All that needs to be changed is the CPU speed.

```
void SetI2CDelay( unsigned int a_msTimeDelay )
{
      unsigned short loop_index     = 0;
      unsigned short ms_index       = 0;
      unsigned short us_index       = 0;

      /*Users can replace the following section of code by their own*/
      for( ms_index = 0; ms_index < a_msTimeDelay; ms_index++)
      {
            /*Loop 1000 times to produce the milliseconds delay*/
            for (us_index = 0; us_index < 1000; us_index++)
            { /*each loop should delay for 1 microsecond or more.*/
                  loop_index = 0;
                  do {
                        /*The NOP fakes the optimizer out so that it
doesn't toss out the loop code entirely*/
                        __asm NOP
                  }while (loop_index++ < ((g_usCpu_Frequency/8)+(+
((g_usCpu_Frequency % 8) ? 1 : 0))));/*use do loop to force at least
one loop*/
            }
      }
}
```

Choosing the File-Based or EPROM-Based Version

To generate a PROM-based I^{2}C Embedded, select the “Generate HEX (.c) File” option in the Deployment Tool Step 2 of 4 dialog box.The programming engines of the file-based and PROM-based processors are identical in the way they handle the VME commands. Their difference lies in the way they interface with I^{2}C Embedded data. For a convenient demo, the file-based version assigns a file pointer to the binary I^{2}C Embedded file directly. The pointer is assigned based on a command line argument. With some minor modification, this version is useful for embedded high-level 32-bit microprocessors that can dynamically allocate RAM and have large amounts of data and code memory. For more modest embedded systems or smaller processors, the PROM-based version is useful because the memory resources are completely defined when compiling the executable.The I^{2}C Embedded file is converted to one or more C files and a header file that are compiled with the core routines.

Programming Considerations for SSPIEM and I2CEM modification with Aardvark I2C APIs

Aardvark is a SPI/I2C adapter which can be used for programming of Lattice FPGA devices with Slave SPI or Slave I2C. Lattice Radiant provides I2CEM example source codes which are modified with Aardvark I2C APIs respectively. However we do not guarantee that these APIs will be supported for all the programming modes incorporated in the .iea files generated by the Lattice Deployment Tool, which are used by our I2CEM source codes. This is due to the limitation of the Aardvark adapter and with its associated read/write APIs meant for the data transfer between the Lattice's algo interpretation logic and the actual programming hardware driver logic. The Aardvark adapter has a buffer limitation of 4 KB and any algo file data above 4 KB will overflow the buffer and will result in a programming failure.

The Deployment tool modes which are effected due to this are the Fast Programming modes for any device, for example the CrossLink-NX device supports Fast Programming mode but will not program with Aardvark APIs. As the Fast Programming mode results in an algo file in which the whole data is passed at once as a whole for Fast Programming and overflows in the Aardvark buffer resulting in a programming failure. The supported programming modes are “Erase Program Verify,” “Background Erase Program Verify,” “Flash Program,” and “SSPI Program.”

The example source code using FTDI can be used to program devices in Fast Programming mode as we guarantee that our drivers work with this mode and the buffer in the FTDI device is large enough to hold large Fast Programming mode data.

Index

Revision History

The following table gives the revision history for this document.

Date	Version	Description
11/072019	2.0	Moved content from Appendix B of “Radiant Software User Guide.” Updated for Radiant software v2.0.
02/08/2018	1.0	Initial release, as Appendix B of “Radiant Software User Guide.”