Programming Xilinx XC9500 CPLDs on IFR 4200 Series Testers

Preface

Introduction

Creating SVF Files

Xilinx ISP Modules

Reference Material

Revision 1.0 *September, 1998* *Printed in U.S.A.*

Preface

About This Manual

This manual describes how to program Xilinx XC9500 CPLDs on IFR 4200 series testers.

Before using this manual, you should be familiar with the operations that are common to all Xilinx's software tools: how to bring up the system, select a tool for use, specify operations, and manage design data.

Manual Contents

This manual covers the following topics.

- Chapter 1, "Introduction," lays out the basic procedure for programming an XC9500 CPLD in an IFR 4200 test environment.
- Chapter 2, "Creating SVF Files," discusses how to create an SVF files on PCs, and on Sun and HP workstations.
- Chapter 3, "Xilinx ISP Modules," describes the In-System programming of Xilinx XC9500 devices on a 4200 series tester.
- Chapter 4, "Reference Material" contains information about **`.avf`** and **`.bvf`** files, and the VPROG module.

Conventions

In this manual the following conventions are used for syntax clarification and command line entries.

- `Courier font` indicates messages, prompts, and program files that the system displays, as shown in the following example.

  `speed grade: -100`

- **`Courier bold`** indicates literal commands that you must enter in a syntax statement.

  `rpt_del_net=`

- *Italic font* indicates variables in a syntax statement. See also, other conventions used on the following page.

  `xdelay` ***design***

- Square brackets "[ ]" indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  `xdelay [`***option***`]` ***design***

- Braces "{ }" enclose a list of items from which you choose one or more.

  `xnfprep` ***designname*** **`ignore_rlocs={true|false}`**

- A vertical bar " | " separates items in a list of choices.

  `symbol` ***editor*** `[bus|pins]`

Other conventions used in this manual include the following.

- *Italic font* indicates references to manuals, as shown in the following example.

  See the *Development System Reference Guide* for more information.

- *Italic font* indicates emphasis in body text.

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets *are not* connected.

- A vertical ellipsis indicates repetitive material that has been omitted.

```
IOB #1: Name = QOUT'
IOB #2: Name = CLKIN'
.
.
.
```

- A horizontal ellipsis "..." indicates that the preceding can be repeated one or more times.

  `allow block` ***blockname loc1 loc2*** `...` ***locn*** `;`

Chapter 1

Introduction

Process Flow

Xilinx XC9500 series parts can be programmed on IFR 4200 series testers using SVF (serial vector files – .svf) generated from JTAG Programmer.

For the 4200 Series the vector files have to be converted to a form that is usable by the system (a binary vector file – .bvf). This .bvf file can then be utilised by a Test Description Language (TDL) module in Computer Aided Program Generation (CAPG) to produce an appropriate MTL test.

This manual describes the operations required to program ISP devices using the following:

- From JTAG Programmer.

  - Generate an erase SVF file and a program/verify SVF file.

- From IFR Ltd.

  -File conversion software (PC executables)

  –MTL and TDL modules for ISP

  –CAPG/XCAPG

  –4200 series ATE

It is assumed that the XC9500 series devices are being programmed on the printed circuit board under test on the 4200 series ATE, and that, therefore, there is an associated test fixture available.

Figure 1 illustrates the process.

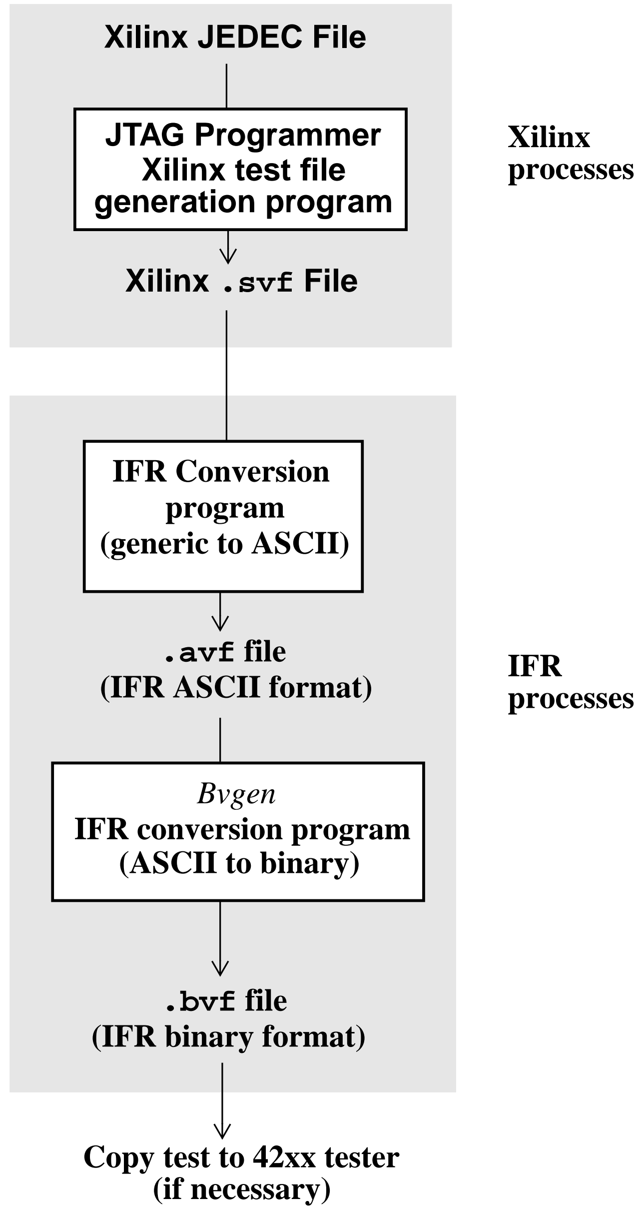


Figure 1-1 Processing ISP vectors (general)

The reason for the two stage conversion process is that ASCII files are easily transferred between different systems and can be easily edited, if required. Binary files, on the other hand contain the same informa-

tion but in a format more suitable for machine access thus providing the necessary speed for a production environment.

It is good policy to carry out two complete programming procedures. The first would use a JEDEC file containing information for programming test vectors, the results of which would then be tested. The second would use a JEDEC file for programming real data.

Documentation

Reference should be made to the following manuals for further details on other, relevant aspects as follows:

- **CAPG**--There are two CAPG manuals, user and reference; both are contained in H4200, CAPG:

  Use the *CAPG User Manual* for information on partitioning and using TDL modules.

  Use the *CAPG Reference* for file formatting information and other reference data. This manual also explains MBNF which is the file description language used in Chapter 4 of this manual.

- **MTL**--There are a number of manuals concerned with MTL. These are contained in H4200, Cover 2. In particular H4200, Cover 2B contains the two MTL Refer-ence manuals which give information on the language itself.
- Xilinx application note XAPP067, *Using Serial Vector Format Files to Program XC9500 Devices In-System on Automatic Test Equipment and Third Party Tools.*
- Serial vector format specification, revision B (produced by Asset InterTech Inc.)

Chapter 2

Creating SVF Files

Creating an SVF File Using JTAG Programmer

This procedure describes how to create an SVF file; it assumes that you are using Xilinx Foundation or Alliance Series software, Version 1.3 or newer. These software packages include the Xilinx CPLD fitter and JTAG Programmer software. JTAG Programmer is available free of charge on the Xilinx World Wide Web site, www.xilinx.com.

JTAG Programmer is supplied with both graphical and batch user interfaces. The batch user interface executable name is `jtagprog`; and the graphical user interface is named `jtagpgmr`. The graphical tool can be launched from the Design Manager or Project Manager, but may also be launched by opening a shell and invoking `jtag-pgmr`. The batch tool is available by opening a shell and invoking `jtagprog` on the command line.

The goal of the following procedure is to create two separate SVF files for each device being programmed. We will show you how to do this using both the batch and the GUI tool. One SVF file contains erase information for the device, another the program verification information for the device, and the third contains verification information. On XC9500 devices the erase vectors should have a 2 ms TCK period.

Using the Batch Download Tool to Generate SVF Files

1. Run your design through the Xilinx fitter and create a JEDEC programming file. You may already have been provided with a JEDEC file; if so, proceed to the next step.
2. Invoke the batch JTAG Programmer tool from the command line in a new shell.

```
jtagprog -svf
```

The following messages will appear:

```
JTAGProgrammer: version <Version Number>
Copyright:1991-1998

Sizing system available memory...done.

***SVF GENERATION MODE***

[JTAGProgrammer::(1)]>
```

3. Set up the device types and assign design names by typing the following command sequence at the JTAG Programmer prompt:

   ```
   part deviceType1:designName1 deviceType2:designName2
   ... deviceTypeN:designNameN
   ```

   where *devicetype* is the name of the BSDL file for that device and *designName* is the name of the design to translate into SVF. Multiple *deviceType*:*designName* pairs are separated by spaces. For example:

   ```
   part xc95108:abc12 xc95216:ww133
   ```

   The **part** command defines the composition and ordering of the boundary-scan chain. The devices are arranged with the first device specified being the first to receive TDI information and the last device specified being the one to provide the final TDO data.

Note: For any non-XC9500(XL) device in the boundary-scan chain, make certain that the BSDL file is available either in the XILINX variable data directory, or by specifying the complete path information in the *deviceType*. The *designName* in this case can be any arbitrary name.

4. Execute the required boundary-scan or ISP operation in JTAG Programmer.

   - **erase** [**-fh**] *designName* -- generates an SVF file to describe the boundary-scan sequence to erase the specified part. The **-f** flag generates an erase sequence that overrides write protection on devices. The **-h** flag indicates that all other parts (other than the specified *designName*) in the boundary-scan chain should be held in the HIGHZ state during the erase operation. Xilinx recommends **erase -f -h** *designName*.
   - **verify** [**-h**] *designName* [**-j** *jedecFileName*] -- generates an SVF file to describe the boundary-scan sequence to read back

the device contents and compare it against the contents of the specified JEDEC file. The JEDEC file defaults to be the *designName*.**jed** in the current directory, or may be alternatively specified using the **-j** flag. The **-h** flag is used to specify that all other parts (other than the specified *designName*) in the boundary-scan chain should be held in the HIGHZ state during the verify operation. Xilinx recommends **verify -h** *designName*.

- **program** [**-bh**] *designName* **-j** [*jedecFileName*] -- generates an SVF file to describe the boundary-scan sequence to program the device using the programming data in the specified JEDEC file. The JEDEC file defaults to be *designName*.**jed** in the current directory, or may be alternatively specified using the **-j** flag. The **-h** flag is used to specify that all other parts (other than the specified *designName*) in the boundary scan chain should be held in the HIGHZ state during the programming operation. The **-b** flag indicated the programming operations should erase the device. This is useful when programming devices shipped from the factory. Xilinx recommends **program -b -h** *designName*.
- **partinfo** [**-h**] **-idcode** *designName* -- generates an SVF file to describe the boundary-scan sequence to read back the 32 bit hard-coded device IDCODE. The **-h** flag is used to specify that all other parts (other than the specified *designName*) in the boundary scan chain should be held in the HIGHZ state during the IDCODE operation.This operation can be performed in any combination of the three SVF files.
- **partinfo** [**-h**] **-signature** *designName* -- generates an SVF file to describe the boundary-scan sequence to read back the 32 bit user-programmed device USERCODE. The **-h** flag is used to specify that all other parts (other than the specified *designName*) in the boundary scan chain should be held in the HIGHZ state during the USERCODE operation. This operation can be performed in any combination of the SVF files.

5. Exit JTAG Programmer by entering the following command:

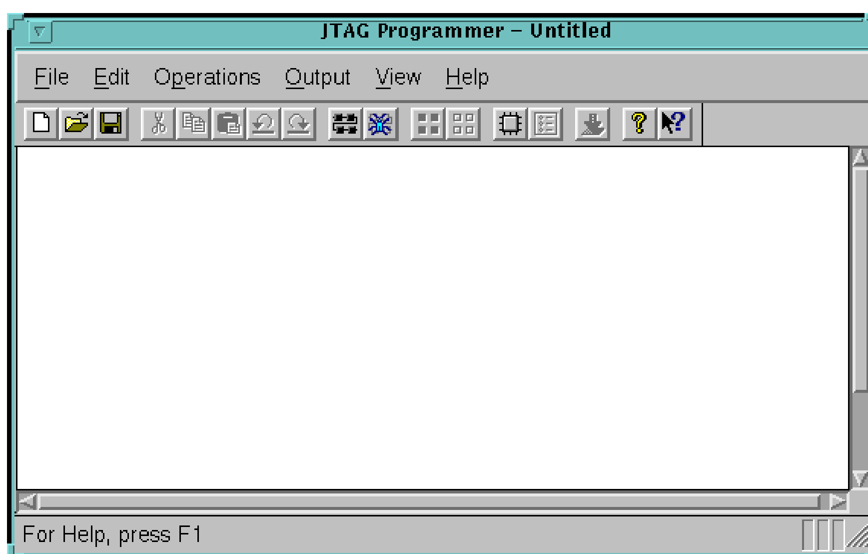
   ```
   quit
   ```

Note: The SVF file will be named *designName*.**svf** and will be created in the current working directory. Consecutive operations on the same *designName* will append to the SVF file. To create SVF files with sepa-

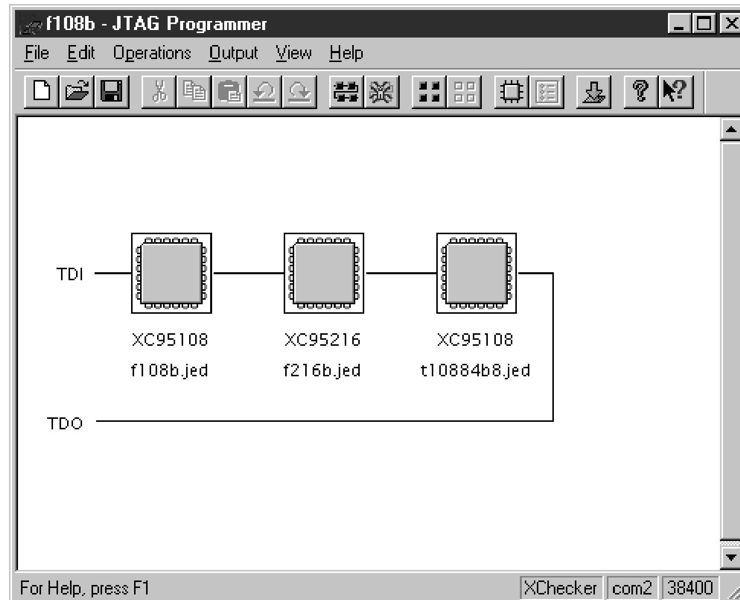
rate operations in each, you will need to rename the SVF file after each operation by exiting to the system shell.

Using the Graphical User Interface to Generate SVF Files

1. Run your design through the Xilinx fitter and create a JEDEC file (you may already have been provided with one).
2. Double-click on the JTAG Programmer icon or open a shell and type **`jtagpgmr`**. The JTAG Programmer will appear.



3. Instantiate your boundary-scan chain. There are two ways to do this. The first is to manually add each device in the correct boundary-scan order from system TDI to system TDO.
   a) Selecting **`Edit`** → **`Add Device`** for each device in the boundary-scan chain.
   b) Fill in the device properties dialog to identify the JEDEC (if it is an XC9500 device) or BSDL (if it is not an XC9500 device) file associated with the device you are adding.



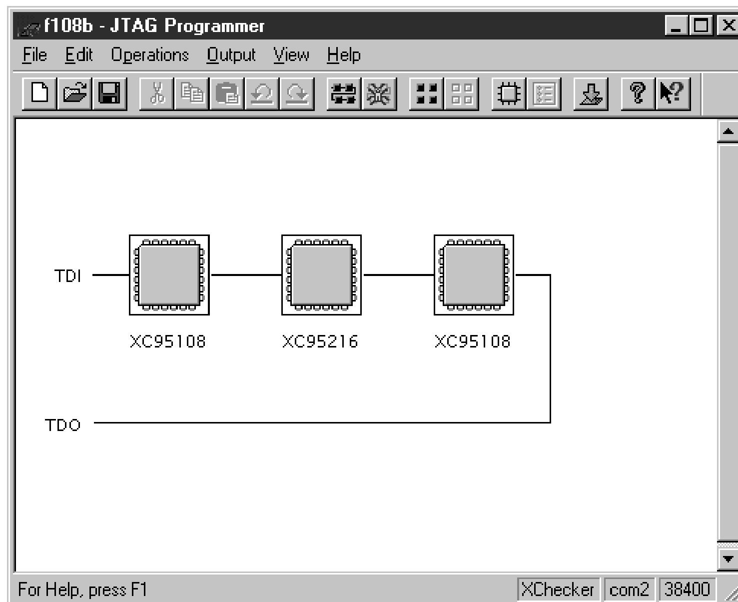
The device type and JEDEC file name will appear below the added device.

The second method is to allow JTAG Programmer to query the boundary-scan chain for devices, and then fill in the JEDEC and BSDL file information. This method will work only when you have the target system connected to your computer with a Xilinx serial or parallel cable. The cable must be powered up by the board under test. The steps are as follows:

a) Initialize the chain as follows:

`File → Initialize Chain`

JTAG Programmer will display the boundary-scan chain configuration as shown:



b) For each device in the resulting chain, double-click on the chip icon to bring up the device properties dialog, then select the JEDEC or BSDL file associated with that device.

4. Put the JTAG Programmer into SVF mode by selecting

   `Output → Create SVF File...`

   to create a new SVF file, or

   `Output → Append to SVF File...`

   to append to an existing SVF file. Fill in the SVF file dialog with the desired name of the target SVF file to be created.

Note: Once you enter SVF mode the composition of the boundary-scan chain cannot be edited in order to ensure consistency of the boundary-scan data in the SVF file.

5. Highlight one of the devices by clicking it once with the mouse. Then, select any of the enable operations from the Operations pull down menu to generate an SVF file to describe the boundary-scan sequence to accomplish the requested operation.

6. When you completed the required operations you may exit JTAG Programmer by selecting:

   `File → Exit`

Note: You may select `Use HIGHZ instead of BYPASS` from the `File` → `Preferences...` dialog to specify that all other parts (not the device selected) in the boundary-scan chain will be held in HIGHZ state during the requested operation.

Note: To generate separate SVF files for each operation you will have to perform the following steps between operations:

a) Select `Output` → `Use Cable...`

b) On the `Cable Communications` dialog box select `Cancel`

c) Select `Output` → `Create SVF File..`

d) Choose a new SVF file and proceed normally.

Chapter 3

Xilinx ISP Modules

Introduction

This chapter describes the methodology for in-system programming of Xilinx XC9500 series devices on an IFR 4200 series tester. XC9500 devices use a standard 4-wire Test Access Port (TAP) for In-System Programming (ISP) and IEEE1149.1 boundary scan (JTAG) testing.

Xilinx XC9500 series in-system programmable devices are supported via test vectors produced by the Xilinx JTAG Programmer software which produces a Serial vector format (SVF) file. JTAG Programmer only uses a subset of the serial vector format. You should refer to the Xilinx documentation (e.g. application note XAPP067) for information on the process of using serial vector format files to program XC9500 devices In-System on automatic test equipment and third party tools.

You should have created two serial vector files: one containing erase vectors for the device, the other containing programming and verification vectors. The erase vectors must be run no faster than 500 Hz, a 2 ms TCK period.

The serial vector file you created must be converted for use by IFR software such as CAPG. This is a two stage process:

1. Convert the SVF file to IFR's ASCII vector file format using the IFR supplied converter. Note that this converter will only support the subset of SVF used by the Xilinx JTAG Programmer software.
2. Convert the ASCII vector file into a binary vector file using the IFR supplied converter.

Note: The supplied IFR translators are PC-based tools appropriate for Win 95/98 and Win NT systems.

Having converted the Xilinx generic test vectors to binary IFR format (.bvf file), CAPG can be run and the program generated, as described in this section.

The process is illustrated in Fig. 3-1:

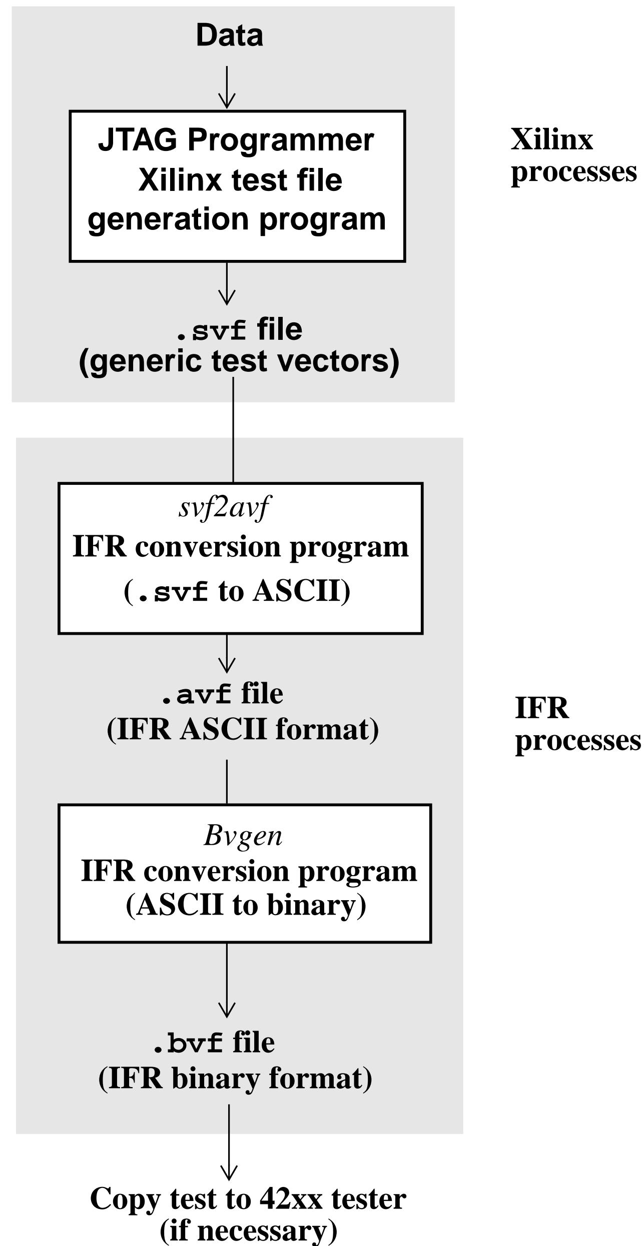


Figure 3-1 Processing ISP vectors (Xilinx)

Running the converters

Each converter is run from the command line, so you should open a system shell on your PC.

Before you can proceed, the vector file obtained by running the *JTAG Programmer* program must be copied to the working directory.

The following ISP software should have already been installed on the CAPG host in the appropriate locations:

- *svf2avf* – Serial vector file to IFR ASCII vectors file converter.
- *bvgen* – ASCII to binary vectors file converter.

For further information on each converter's format, please see IFR Ltd's documentation.

svf2avf (Serial vector file to ASCII vector file)

svf2avf must be run on the same platform as CAPG/XCAPG.
To run it, type svf2avf, followed by a set of arguments according to the format:

```
svf2avf[-d] <input_file> <output_file>
```

where:

-**d** is an optional switch which allows you to include comments in the output file – to aid readability.

input_file is the name of the serial vector file to be converted.

output_file is the name of the ASCII vector file to be output. This will be a **.avf** file ready for conversion to a **.bvf** file (see next section).

The converter takes a Xilinx Serial Vector File and converts it into an IFR ASCII vector file. The converter only supports a subset of SVF as used by the Xilinx *JTAG Programmer* software. The following svf statements are recognised:

```
SDR, SIR, RUNTEST, TRST, ENDIR, ENDDR, STATE
```

The resulting .avf file has SOURCE specified as 'JTAG Programmer'.

The converter takes into account the requirement to repeat blocks of vectors that correspond to failing SDR statements. The error recovery loop is defined in the Xilinx application note XAPP067. The converter uses .avf BREAK statements in appropriate places to identify blocks of vectors that require error recovery.

A block of vectors that requires error recovery is given a block id of ‘1’ all other blocks are given a block id of ‘0’.

bvgen (ASCII vector file to binary vector file)

bvgen must be run on the same platform as CAPG/XCAPG.
To run it, type bvgen, followed by a set of arguments according to the format:

bvgen<*input_file***> <***output_file***>**

where :

input_file is the name of the ASCII vector file **.avf** to be converted

output_file is the name of the binary vector file **.bvf** to be output

Having entered this command line, you should now have a .bvf file to use during MTL runtime for programming the ISP devices.

The CAPG phase

This phase involves use of the partitioning toolkit (see CAPG manual). You must generate a partition for each ISP port that includes all the devices to be programmed by that port. Once a partition has been created, you add an entry in the device library for that partition. You can use **SVF.tdl** in the IFR device library as a framework for creating your library entry. You must also specify the SVF TDL module as the test to use for that partition.

The only PIOs (Primary Input Outputs) are the four TAP pins and it is important that these be appropriately named tms, tck, tdi, and tdo. Names have to match those in the pinlist and TDL module.

The class for the partition library entry is also important as it determines where in the final MTL program the partition is programmed. The classes are mosp1 or mosp2. If the class chosen is mosp1, the partition is programmed before digital tests. If the class is mosp2 programming occurs after digital tests.

If you are doing two programming runs (see Chapter 1) you would choose `mosp1` first, program test vectors and test the results. You would then change the SVF file input and this time, at this point, select `mosp2` to program real data.

Having selected the class for the partition library you can now run the SVF module. During running of the module MTL test code is generated. If you have carried out the CAPG phase on the target tester, you can run the final MTL program. If not, you will have to transfer the relevant files.

MTL runtime

When you run the generated MTL program, Vprog is called. This controls loading of the .bvf file into the pattern generator and programming/verification of the ISP devices on the UUT.

`Vprog` handles

- Loading of each block of data into the pattern RAM.
- Guarantees the required delay between blocks.
- Configures testpoints to the required line of the pattern generator.
- Configures the mask of the sdo pin.
- Patches MTL repeat instruction in Master Test Controller (MTC) control RAM with required repeat count for the block loaded.
- Flags when no more blocks remain to be processed.
- Handles printing and logging of results

The `Vprog` functions are defined in Chapter 4.

The MTL program to control programming of the device will consist of some initiation code and calls to `Vprog`, followed by a repeat loop. The following lines illustrate this.

```
1  configure testpoints/open binary file
2  configure mask
3  load a block of vectors
4  if no more vectors stop
5  repeat for number of vector in block
```

```
6   apply a vector
7   end repeat
8   check block ok
9   if block failed attempt error recovery if required
10   goto 3
```

Some blocks of vectors require error recovery if they fail. These blocks correspond to SDR instructions in the **`.svf`** file. Refer to XAPP067 for information on the error recovery process.
Briefly, error recovery consists of a delay and then attempting to apply the block again. The error recovery loop is attempted 32 times before the block is considered to fail.

Chapter 4

Reference Material

File formats

The information given in this chapter is intended to describe those files which are produced by means of IFR Ltd's software. The descriptions are in MBNF, the use and structure of which is explained in the *CAPG Reference manual,* and also in the *MTL Reference – Core manual.*

ASCII vector file (.avf)

The following MBNF specifies the format of the ASCII vector file used as input for ISP programming of Xilinx ISP devices on a 4200 series tester.

```
$$ mi_avf      = {source | rate | vector | break}{comment|}
$  source      = "SOURCE" from
$  from        = "JTAGProgrammer"
$  rate        = "RATE" time
$  vector      = pin_data pin_data pin_data pin_data pin_data
$  pin_data    = ["." | "X" | "0" | "1" | "H" | "L"]
$  break       = "BREAK" time [block_id||]
$  block_id    = [<alpha> || <digit>]
$  time        = <integer> units
$  units       = ["f" | "p" | "n" | "u" | "m" | "R" | "k" |
                 "K" | "M" | "G" | "T"]
$  comment     = "#" text
$  text        = (* all remaining characters up to a newline *)
```

All vectors are 5 characters wide, each character corresponds to one of the isp programming pins and is ordered as follows:

Maker	Ch. 1	Ch. 2	Ch. 3	Ch. 4	Ch. 5
Xilinx	Not used	TMS	TCK	TDI	TDO

RATE specifies the rate at which vectors should be applied by the tester and should appear only once in the ASCII vector file.

Binary vector file (.bvf)

The following MBNF specifies the format of the binary vector file used as input for ISP programming of ISP devices on a 4200 series tester.

```
$$ mi_bvf       = magic_number cycle_time {block}
$  magic_number = <posint> (* byte 0..255 *)
$  cycle_time   = <posint> (* long 0..4294967295, time in nanosec *)
$  block        = header data
$  header       = sync_byte block_id no_vectors delay
$  sync_byte    = <posint> (* byte 0..255 *)
$  block_id     = <posint> (* byte 0..255 *)
$  no_vectors   = <posint> (* word 0..65535 *)
$  delay        = <posint> (* word 0..65535 *)
$  data         = {<posint>} (* no_vectors x bytes *)
```

`magic_number` identifies the file as a valid binary vector file and gives the possibility of identifying different version/variants of the file. For binary files containing Xilinx In-System programming vectors it is 0x20.

`cycle_time` specifies the rate, in nanoseconds, at which vectors should be applied.

`sync_byte` at the start of every block of data allows the file to be checked for corruption.

`block_id` provides a mechanism of identifying different blocks of vectors. For instance, this is used to identify when a device id check is being performed.

`no_vectors` gives the number of bytes of vector data that follow the header.

`delay` is in milliseconds and states the minimum amount of time that must pass between applying the vectors contained in the current block and applying the vectors of the next block.

`data` --Each byte of data contains 8 bits of information to load into the pattern ram on the 4200 Series tester. No block can consist of more than 8192 bytes of data i.e. consist of more than 8k vectors. This is dictated by the depth of the pattern ram on the 4200 Series.

VPROG Module

The Vprog MTL module contains functions to control the loading and processing of the binary vector file.

Vprog provides the following functions –

```
Vprog.svf.init(file string ,p1,p2,p3,p4,p5,p6,p7,p8 pin) void
```

Configures each of the testpoints p1 to p8 to a line of the pattern generator. It locates the repeat instruction in MTC control ram that will require patching with the correct repeat count. Opens the binary vector file.

```
Vprog.svf.mask(masktp, montp, sdo pin)void
```

Configures a mask pin, used to mask monitors in the pattern ram on vectors when a monitor is not required.

```
Vprog.svf.data(reset int)void
```

When reset is 0 it opens the binary vector file, otherwise it reads the next block of data and performs any delay required by the previous block.

With Vprog.svf.data, when the end of the binary vector file is encountered it sets MTL FLAG(1) to indicate end of file. If a block of vectors requires error recovery it sets MTL FLAG(2).

```
Vprog.svf.errorcount(maxretry int)void
```

Checks to see if the current block has been re-tried more than a specified number of times. If a block has been re-tried more than the specified number of times the MTL FLAG(2) is cleared to indicate that the block does not require re-trying.

```
Vprog.svf.errordelay()void
```

Delays the amount of time required by the error recovery loop.

```
Vprog.svf.cleanup()void
```

Called to close bvf file at end of a test.

```
Vprog.svf.scan_func()void
```

Replaces Test.Onscan function. `VPROG.SVF.scan_func ()` stores the number of failures.

`Vprog.svf.display_func()void`

Replaces `Test.Display.Function` and handles display of results.

`Vprog.svf.print_func()void`

Replaces `Test.Print.Function` and handles the printing of results.

`Vprog.svf.log_func()void`

Replaces `Test.Log.Function` and handles the logging of results.

`Vprog.delaytweak (mul float)`

Increase or decrease delays by the specified multiplier. This is sometimes useful when debugging and the manufacturer's specified delay needs to be adjusted. This function must be called after `Vprog.svf.init()`