JTAG logic configuration utilities

Development Systems

Document number:	JTAGPROG-0000-PRIV-DOC-D
Date of Issue:	29 August, 2001
Author:	Trevor Howlett
Authorised by:	

Abstract

This document describes a suite of utilities that have been written to allow programming of configurable logic and flash memory devices on development boards

Keywords

JTAG, Multi-ICE, Programming, Configuration, Download, Flash, FPGA, PLD

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

1.1.2 Change history

Issue	Date	By	Change
A01	22 July, 1999	Jonathan Travers	First draft
A02	29 July, 1999	Jonathan Travers	Updated for 1.81, added revision history
A	3 August, 1999	Jonathan Travers	Minor changes and additions
B	14 August, 2000	Trevor Howlett	Updated for 2.00
C	14 May, 2001	Trevor Howlett	Updated for 2.02
D	29 August, 2001	Trevor Howlett	Updated for 2.10

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning

2 INTRODUCTION

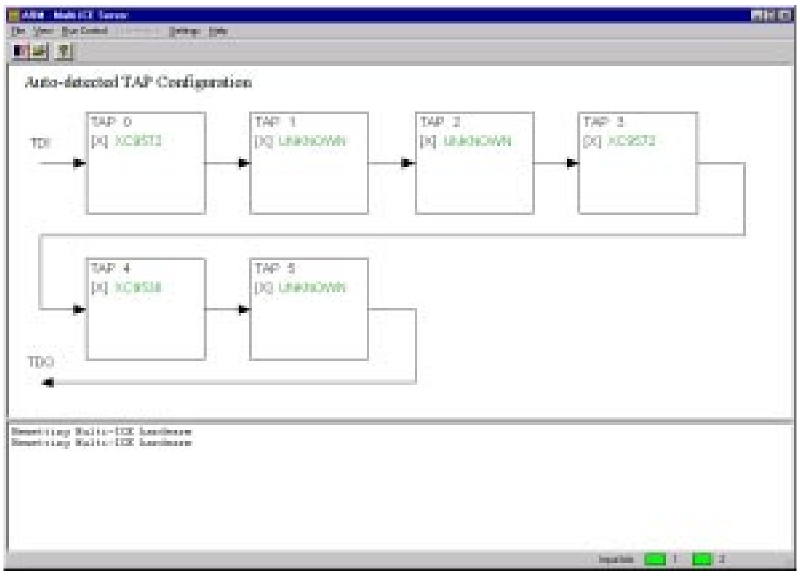
This document describes a suite of utilities that have been written to allow programming of configurable logic and flash memory devices on development boards. All the utilities communicate with the devices using JTAG, through ARM's Multi-ICE interface unit. The utilities are intended to provide a complete in-system programming solution, capable of directly programming common devices and also writing configurations to flash memories that are used to configure logic devices on power-up.

2.1 Supported devices

- **FPGAs** – These are large configurable logic devices. The devices are based on SRAM technology and are therefore volatile. Configurations can be loaded directly using JTAG – the utilities currently support Xilinx 4000 series, Virtex FPGAs.
- **PLDs** – These are smaller, faster configurable logic devices. They can also be programmed using JTAG, but are based on non-volatile FLASH technology and will thus retain their configuration permanently. The utilities support Xilinx 9500 and 9500XL devices, and also some Philips PLDs.
- **Flash memory** – FPGAs on development boards are usually configured on power-up from some sort of non-volatile memory device. Utilities are provided for programming Intel and Atmel flash devices that can be used for this purpose. These devices do not have JTAG interfaces, so indirect programming methods have been devised that make use of JTAG-capable devices which are connected to the flash chips.

2.2 Batch programming

Since there may be several different devices on any particular development board, a utility (ProgCards) has been written that supports a flexible scripting system for programming entire sets of devices. All the programming methods are supported, and the utility also provides a configurable menu system if support for several different configurations is required.



```
Resetting Multi-ICE hardware
Resetting Multi-ICE hardware
```

Example of a configured Multi-ICE server

2.3 Multi-ICE

All the utilities communicate with devices using JTAG via ARM's Multi-ICE interface unit. In order for the utilities to work, the Multi-ICE server must be running and correctly configured. The server window should show a diagram of the JTAG devices present on the board (see example above). Multi-ICE is capable of auto-detecting most of the devices supported by the programming suite

Note 1: The server does not have to be running on the same PC as the programming utilities. They all support remote connection by specifying an optional server name parameter.

Note 2: The programming utilities require Multi-ICE release 1.4 or above.

2.4 The utilities

Six executables are supplied in the programming suite:–

- **FPGA** – This utility downloads temporary configurations to Xilinx FPGAs. It is documented in section 3.
- **PLD** – This utility downloads data from SVF files to PLD devices. It is documented in section 4.
- **BSFlash** – This utility programs flash devices via boundary scan on a connected device. It is documented in section 5
- **ChainTest** – This utility analyses boundary scan chains. It is intended for use when writing description files for BSFlash. See Section 5.8 for more details.
- **ProgCards** – This utility provides a scripting engine for all the programming methods described here. The complete functionality is present in the utility – it does not require the other utilities to run. See section 6 for more information on ProgCards.

2.5 Version history

The following table summarises the versions of the programming suite that have been released to date. All the utilities should report a version number when they are run:-

Date Released	Version Number	Notes
Before April 1999	No version number reported	The initial releases of the programming suite supported only Xilinx 95xx PLDs, Xilinx 4000 FPGAs and one indirect flash programming algorithm.
9 April 1999	Version 1.5	This release contained performance improvements (using new Multi-ICE features).
9 June 1999	Version 1.6	This release contained the new boundary scan programming utility, BSFlash. Minor changes were also made to the PLD programmer.
-	Version 1.7	This version was for development purposes only and was not released. If you have a utility that reports this version number, please replace it with an officially released version.
23 July 1999	Version 1.8	Support for a wide range of new devices, including Xilinx Virtex FPGAs, Xilinx 95xxXL PLDs and a new indirect flash programming method (for Intel devices). BSFlash is now integrated into ProgCards.
3 August 1999	Version 1.81	Added support for wildcards and lists of devices in ProgCards board files.
20 January 2000	Version 1.85	Support added for Intel MCS type.
15 February 2000	Version 1.87	Multi-ICE server version check fixed, programming PLDs with Multi-ICE release 1.3 would fail with Multi-ICE error 38. Support added for TAP controllers with more than one device connected to it.
14 August 2000	Version 2.00	Support added to progcards for programming Altera based logic modules with .rbf images (intelflash/intelflashverify extension). Board files (.brd) used with this release can now use TAPs numbered 0..n-1 instead of devices numbered 1..n. PLD programming enhanced to work with Altera 7000AE family.
14 May 2001	Version 2.02	Support added for Logic Modules using Intel Strata Flash, and changes to allow compatibility with Multi-ICE 2.1.
29 August 2001	Version 2.10	Support added for programming of the AP motherboard system flash. Also StepnDisplayText and StepnPause added to board file commands.

3 FPGA PROGRAMMING (XILINX)

3.1 Description

The ‘FPGA’ utility is provided for downloading temporary configurations to Xilinx FPGAs. It supports all 4000 series and Virtex / Virtex E FPGAs. Configurations downloaded with this program are temporary – they will be lost if the FPGA PROG pin goes low, or the power is removed.

3.2 Syntax

```
FPGA <filename> <chainpos> <IR length> [server]
```

- **Filename** – specifies the location of the bit file containing the configuration data. These files are generated by the Xilinx place and route tools. The program will also accept MCS format files (used to program flash and PROM devices), but will only work with files containing single images stored in ascending order, beginning at address zero.
- **Chainpos** – specifies the location of the FPGA device in the Multi-ICE scan chain (note that devices are numbered from zero)
- **IR Length** – specifies the JTAG instruction register length for the FPGA, in bits. This is used to distinguish between 4000-series (IR length = 3) and Virtex FPGAs (IR length = 5).
- **Server** – (optional) specifies the name of the computer on which the Multi-ICE server is running. If omitted, it is assumed to be running on the same computer as the utility.

3.3 Performance

Programming time will depend on the size of the device, and the speed of your PC. The program will report download throughput in kilobytes per second, and the percentage of the file that has been downloaded. For Virtex FPGAs, the number of configuration frames downloaded is also displayed. In general, a single FPGA download should take approximately 15 seconds.

3.4 Notes

3.4.1 4000-series FPGAs

- In order to program 4000-series FPGAs, the program needs to be able to control the FPGA’s PROG and INIT pins. PROG is used to reset and clear the FPGA, and INIT is held low to prevent the device loading a configuration from any attached memory device. The program assumes that Multi-ICE’s nTRST output is connected to the PROG pin, and the nSRST output is connected to the INIT pin. Without these connections, programming will probably not be successful.
- In 4000-series FPGAs, the JTAG pins are available as user IO after configuration. Unless you have instantiated the BSCAN symbol in your design, JTAG will not be functional after the device configures, because the scan chain will be broken at the FPGA. It is recommended that you instantiate the BSCAN symbol in all your designs, and do not use the JTAG pins as IO. If you do not include the BSCAN symbol, the startup sequence order in the bitgen options becomes very important. You must make sure that making outputs active is the last step in the startup sequence. If it is not, the steps of the sequence after the outputs are made active will not take place because they are clocked by TCK, which is disconnected as soon as the design becomes active. Note that by default, making outputs active is not the last step in the sequence.

3.4.2 Virtex and Virtex E FPGAs

- Control of PROG and INIT is not required for programming of Virtex FPGAs, although the program will still assert (take low) nTRST before programming begins. It is advised that nTRST and nSRST are connected as for 4000-series devices.
- If Multi-ICE's nTRST signal is not connected to a Virtex device's PROG pin, the DONE output from the FPGA will remain high during programming. Download will still work, but no indication will be available on the board (it is usual for DONE to be connected to an 'FPGA OK' LED).
- In the bitgen options for a Virtex FPGA, there is an option to specify the start-up clock used during configuration. An image written into a configuration PROM will usually have a CCLK start-up clock, but when downloading via JTAG, the start-up clock should be TCK. In order to avoid the need to build multiple bit files for each configuration, the FPGA utility contains code to modify this setting as the bit file is downloaded. Therefore a bit file with CCLK start-up can also be downloaded using the utility.

4 PLD PROGRAMMING

4.1 Description

The ‘PLD’ utility is provided for writing configurations to PLD devices. The utility is in fact just an interpreter for serial vector format (SVF) files. This is an industry standard file format for specifying JTAG operations, designed for exchanging vectors between test equipment. The program has no knowledge of the programming algorithms for different PLD devices – it merely executes the commands in the SVF file produced by the PLD place and route tools. The utility has been tested with files produced for Xilinx 9500 and 9500XL devices, Altera MAX AE devices, and also with some Philips PLDs.

4.2 Syntax

```
PLD <filename> <chainpos> <IR length> [server]
```

- **Filename** – specifies the location of the SVF file containing the configuration data.
- **Chainpos** – specifies the location of the PLD device in the Multi-ICE scan chain (note that devices are numbered from zero)
- **IR Length** – specifies the JTAG instruction register length for the PLD, in bits. Xilinx PLDs have an IR length of 8, Philips PLDs have an IR length of 10.
- **Server** – (optional) specifies the name of the computer on which the Multi-ICE server is running. If omitted, it is assumed to be running on the same computer as the utility.

4.3 Performance

Programming time will depend on the size of the device, and the speed of your PC. Since PLDs are based on flash technology, programming can be quite slow, and there may be significant pauses while the memory is erased. The program will report throughput in kilobytes per second, and the percentage of the file that has been executed. This only represents the position in the SVF file and is not an estimate of the required programming time. SVF files for Xilinx 9500 devices, for example, will cause the percentage to freeze at very low values, because this is the part of the file contains the erase commands.

4.4 Notes

- If you specify an IR length of 8, the program will assume you are programming a Xilinx PLD and change its behaviour in accordance with guidelines from Xilinx. This affects what happens when incorrect data is read back from the device. This will normally cause the program to immediately report an error, however Xilinx have defined a proprietary retry sequence that is executed in this situation. The program will only do this if the IR length is 8 bits.
- The utility will assert nTRST before it begins programming, and will leave nSRST asserted throughout the programming sequence. This behavior is intended to hold any 4000-series FPGAs in a reset state, which guarantees that their JTAG circuitry will be active. See section 3.4.1 for details of recommended connections to Xilinx FPGAs.

5 BOUNDARY SCAN FLASH PROGRAMMING

5.1 Description

BSFlash is a utility for programming flash memory devices using the boundary scan facilities of an attached device. The program scans an EXTEST instruction into the device that is connected to the flash chip, and then uses its boundary scan chain to control the outputs of the device, thus programming the flash.

5.2 Requirements

In order to use the utility, you will need to set up several things:-

- You need to write a Flash Description File. This contains a description of the scan chain in the device you are using, and details of the specific flash parts that are connected. There is more information on the flash description file later in this section.
- You need an image to write into the flash. BSFlash currently supports MCS format files, Xilinx BIT files and raw binary files.

5.3 Syntax

The program has a number of command line options, depending on the required function:–

Program mode – the specified image will be programmed at the given start address:

```
bsflash /D:dscfile /T:TAPpos /B:bitfile [/A:address] [/S:server]
bsflash /D:dscfile /T:TAPpos /R:rawfile [/A:address] [/S:server]
```

Verify mode – If the /V switch is used, the contents of the flash at the given address will be verified against the specified image:

```
bsflash /D:dscfile /T:TAPpos /B:bitfile [/A:address] [/S:server] /V
bsflash /D:dscfile /T:TAPpos /R:rawfile [/A:address] [/S:server] /V
```

Output mode – If the /O switch is used, the contents of the flash will be read out and written to the specified file. You need to supply a start address and the number of bytes to read:

```
bsflash /D:dscfile /T:TAPpos /B:bitfile /A:addr /L:size [/S:server] /O
bsflash /D:dscfile /T:TAPpos /R:rawfile /A:addr /L:size [/S:server] /O
```

Test mode – The /t switch will initiate the special test mode. See later for a description of test mode.

```
bsflash /D:dscfile /T:TAPpos [/S:server] /t
```

5.4 Options

`/D:dscfile`	This specifies the flash description file to use for this device. The parameter is always required (no default).
`/T:TAPpos`	Specifies the TAP controller to use. These are numbered from zero (counting from the left in the Multi-ICE server window). The parameter is required (even if there is only one TAP controller).

`/A:address`	Specifies a start address for an image. The address is given in hexadecimal (with no leading characters and not case sensitive). For example: /A:100000 specifies a start address 1MB into the address space. Note that MCS files contain embedded addresses, which will override any start address that you give.
`/L:size`	Specifies the number of bytes to read in output mode. The parameter is given in decimal. For example /L:1024 will read out 1K of data.
`/S:server`	Specifies the host name of the computer running the Multi-ICE server. This parameter is optional – if absent it will be assumed that the Server is running on the same machine as the program.
`/B:bitfile`	Gives the filename of an MCS or Xilinx BIT file for programming. The program will automatically detect which format is being used. When in output mode, using this parameter will cause an MCS file to be generated from the flash data.
`/R:rawfile`	Specifies the filename of a raw binary file for programming. The file will be copied byte-for-byte into the flash. When in output mode, using this parameter will cause a raw binary image to be generated from the flash data.

5.5 Writing a flash description file

In order to use the programmer, you will need a flash description file for the device you are using. This specifies the location of address, data and control bits in the boundary scan chain. Here is an example file:-

```
[General]
FileType = FlashConfig
Name = SA1100 for Prospector/P1100 board

[Config]
Type = Intel
ChipSize =  8388608
SectorSize = 131072
BlockSize = 32
ManufacturerID = 89
DeviceID = 15
Width = 16
Parallel = 2

[ScanChain]
DRLength = 279
IRLength = 5
Extest = 0
Bit0   = 0:0:0       ROMSEL(I)
Bit1   = 0:1:0       RESETO(O)
Bit2   = 0:0:0       RESET(I)
Bit3   = 0:0:0       TXD3(I)
Bit4   = 0:0:0       TXD3(O)
Bit5   = 0:0:0       TXD3(E)
Bit6   = 0:0:0       RXD3(I)
Bit7   = 0:0:0       RXD3(O)
Bit8   = 0:0:0       RXD3(E)
Bit9   = 0:0:0       TXD2(I)
Bit10  = 0:0:0       TXD2(O)
...etc
```

The **general** section is present to identify this as a valid flash description file (the line `FileType = FlashConfig` must be present or the program will reject the file). The name field simply gives a description for this configuration. It gets printed by BSFlash after it reads the file.

5.5.1 Flash configuration

The config section is used to describe the arrangement of flash chips connected to the device. The type field is used to specify a programming algorithm to use. Currently the only supported algorithm is Intel, which uses the Intel Standard Command Set to program a block-erasable flash. The other fields in this section give algorithm-specific details of the chip sizes and arrangement. BSFlash is able to support most sensible arrangements of flash chips. Parallel configurations are supported, where several chips are present at the same address to make up a wider data bus. Both 8-bit and 16-bit devices may be used, and the program also supports banked arrangements, where several chips are present at consecutive addresses.

`ChipSize`	This specifies the total size, in bytes, of an individual flash chip. The parameter is given in decimal. The example shows an 8MB device.
`SectorSize`	This gives the size, in bytes, of one of the erasable blocks in an individual flash chip. In the example this is 128K. Note that the program always erases devices block-by-block. The chip erase command, if present, will not be used.
`BlockSize`	This gives the size, in bytes, of the write buffer in each flash chip. In the example it is 32 bytes.
`ManufacturerID`	This gives the manufacturer ID of the flash device. The parameter is specified in hexadecimal. The program will check this information each time it accesses a new chip, and report an error if the wrong value is returned.
`DeviceID`	This gives the device ID of the flash device. The parameter is specified in hexadecimal. The program will check this information each time it accesses a new chip, and report an error if the wrong value is returned.
`Width`	This specifies whether the devices are being used in 8-bit or 16-bit mode. The value should be either 8 or 16.
`Parallel`	This specifies how many devices are connected in parallel (in effect – the width of the data bus). Buses of 8, 16 and 32 bits are supported.

After reading this data, BSFlash will make some assumptions which you should be aware of:-

- All the connected flash chips are of the same type, and operating in the same mode.
- Data is transferred across the bus in whatever sized words you have specified. For example if you have specified two 16-bit devices in parallel, data will be transferred one 32-bit word at a time.
- The address bus uses byte addresses. So in the example above, where the word size is 32-bits, the address will be incremented by 4 after each access. Other arrangements are possible through careful remapping of the address pins in the scan chain description (see later).
- The first flash chip (or chips in a parallel system) is located at address 0, and successive chips in the bank are located at consecutive addresses. There is currently no way to specify a different base address for the flash memory. If, for example, your flash begins at address 0x80000 (512K) and your flash chips are 2MB devices, the program will incorrectly assume that the first chip boundary occurs at address 0x200000 (2MB), whereas it is actually at 0x280000. It is likely that the flash chips’ address pins will be directly connected to the JTAG device you are using, in which case chip boundaries will occur in the places the program expects them to. The only situation in which you need to consider this problem is if the connections to the flash chip pass through some sort of address remapping logic.

5.5.2 The Boundary Scan Chain Description

The ScanChain section of the file contains a description of the JTAG boundary scan chain in the device being used to access the flash memory. The section starts with some general fields:-

`DRLength`	This specifies the number of bits in the EXTEST boundary scan data register.
`IRLength`	This specifies the number of bits in the instruction register for this device.
`Extest`	This specifies the bit pattern to scan into the instruction register to put the device in EXTEST mode. Note that according to the IEEE JTAG spec, this should always be 0.

Next there is a description of each bit in the scan chain. The bits are numbered from 0 – and bit0 is the first one that gets scanned in (this is the same numbering scheme as used in BSDL files). An example line looks like this:-

```
Bit29  = 5:3:1       A1(O)
```

The important part of the line is the three numbers separated by colons. BSFlash will ignore anything else on the line and it is recommended you use this space to add a comment that describes the bit. The three numbers are interpreted as follows:–

```
<pin type>:<bit type>:<index>
```

The **pin type** identifies which group of pins this bit belongs to. The choices are as follows:–

- **0** **Set Value** – this bit is not used for flash programming and should be permanently set to 1 or 0 (depending on the bit type value)
- **1** **OE** – this bit controls the output enable pin.
- **2** **WE** – this bit controls the write enable pin.
- **3** **CE** – this bit controls the chip select pin.
- **4** **Data** – this bit controls a data pin.
- **5** **Address** – this bit controls an address pin.
- **6** **Reset** – this bit controls a pin that is connected to the flash reset input.

The **bit type** describes the specific function of this bit. It is common for up to three bits to be assigned to each IO pin in a boundary scan chain. These are generally the output value of the pin, the tristate control for the IO pad, and the captured input value from that pin. Here is a typical example from a PLD scan chain:–

```
Bit87  = 5:5:13   XA13(E)
Bit88  = 5:3:13   XA13(O)
Bit89  = 5:2:13   XA13(I)
```

The choices for bit type are as follows:–

- **0** **Set permanently to zero** – This should only be used if the pin type is 0.
- **1** **Set permanently to one** – This should only be used if the pin type is 0.
- **2** **Input** – Specifies that the bit will contain captured input data from an IO pad. Note that the program currently ignores inputs for anything other than data pins.
- **3** **Output** – Specifies that this bit controls the output value to the IO pad.
- **4** **Inverted output** – Specifies that this bit controls the output value to the IO pad, and that this value should be written to the scan chain inverted.

5 **Enable** – This bit is a tristate control (for an output pad) or a direction control (for a bi-directional pad). Placing a one in the bit will enable the output (or set the pad to be an output) and placing a zero in the bid will tristate (disable) the output (or set the pad to be an input).

6 **Disable** – This is a tristate control, but works the other way round to an enable. Placing a one in the bit will tristate (disable) the output (or set the pad to be an input) and placing a zero in the bit will enable the output (or set the pad to be an output).

The **index** field identifies particular pins, for those pin types which are buses. This includes the data and address pin types. The index is numbered from zero.

5.5.3 Notes

Scan chain length – If the scan chain you are using has 255 bits or less, there are important issues that you should understand about the way BSFlash will try to access the scan chain. Refer to the section on Access Speed for more information.

Inverted control lines – Note that for consistency BSFlash uses an internal active high model for the control lines (OE, WE, CE and RESET). These lines are almost always active low in real systems (nOE, nWE, and so on), so the output bits for these pins will usually be set to type 4 (inverted output).

Note that you are not forced to accurately describe the function of all the pins. In fact most of the bits in the scan chain are likely to be set to a permanent value for one reason or another. (for example – chip enable lines may have to be set correctly to connect the flash to the data bus).

Reset – the reset line is driven active by BSFlash for about one second before programming begins. If this is not the required behaviour, or the signal is not available – simply omit this pin type from the description file and set any reset lines to the desired permanent value. There will be a delay before programming begins, but no outputs will change.

Address lines – It may be necessary to remap the address bits in the boundary scan chain. Remember that BSFlash assumes that the address bus uses byte addressing. You can change this behavior by remapping the address indices. Here are two examples:-

- Imagine a system where a 16-bit flash part is connected to the bottom half of a 32-bit data bus with byte addressing. Because the flash is 16-bits wide, the program will increment the address by 2 bytes after each access, whereas the required behavior is an increment of 4 bytes (to skip the ‘unoccupied’ half of the bus). However, you can force this behavior by assigning a constant zero value to address pin zero and then specifying the other pins as follows: define the real A1 as A0 in the description file, A2 as A1 and so on. This has the effect of multiplying the program’s address by two before outputting it to the bus, which is precisely the required behavior.
- Imagine a system where two 16-bit flash parts are connected in parallel to a 32-bit data bus with word addressing. The program will increment the address by 4 after each access, but the bus is word-addressed, so the required increment is actually 1. The solution is to remap the pins as follows: define the real A0 as A2 in the description file, A1 as A3, A2 as A4 and so on. The bottom two bits of the address will be discarded, effectively dividing it by 4, which is the required behavior.

CE – The chip enable pin is not currently used during programming and will be set permanently active. You should still use pin type 3 in your description file as it may be used in future versions of BSFlash.

Bus enables – Some chips contain tristate bits that control an entire bus (for example – one bit controls the direction of the whole data bus). This is not a problem, since BSFlash does not individually modify the tristate control bits for buses. Specify the bit with the correct pin and bit type, and set the index to zero (or any other value you like)

Unidirectional buses – It is possible that the data bus interface to the device is unidirectional with separate data in and data out buses. This arrangement can be specified quite easily in the description file. Set the input and output bits in the correct places (4:3:x for an output, 4:2:x for an input). If these IO pads also have tristate controls, set them to the appropriate permanent value (0:0:0 or 0:1:0). There may also be a data direction output. Find the output bit for this pin, and set it as a data enable or disable (4:5:0 or 4:6:0). The value output from this pin during programming will then be the desired direction for the bus.

5.6 Test mode

When writing a description file, it is quite likely that mistakes will be made, especially if the boundary scan chain contains hundreds of bits. It is sometimes difficult to determine whether particular pins are working properly. For this reason BSFlash supports a special test mode, which makes debugging the file (and other board problems) a bit easier.

Test mode is initiated by specifying the /t switch on the command line. The program will sit in a loop toggling data bits 0 and 1 continuously (bit 1 will be an inverted version of bit 0). By inserting these output bits into the boundary scan description, you can test whether particular bits are having the desired effect on the pins of the chip.

So for example – to test whether a particular bit actually is the output bit for a particular pin, and that the pin is correctly connected to the board, do the following. Rewrite the description file so that all the bits apart from the one under test are set to permanent values (0:0:0 or 0:1:0). Set the bit under test to be a data output (4:3:0) and then run the test. If everything is working properly, you should see a rapidly changing waveform if you probe the pin on the board. Note that this may not be regularly spaced (no attempt is made by the program to space out the transitions).

5.7 Access speed

BSFlash will output performance statistics as it programs the flash. On a fast machine, the performance may approach 1K/s, but due to the number of JTAG operations that must be performed just to change the state of one bit in the scan chain, this is about as fast as the programming is likely to get.

If your scan chain has 255 or less bits, BSFlash is able to take advantage of faster access commands that are built into the Multi-ICE unit. These commands rely on non-input bits in the scan chain being recirculated correctly. For BSFlash to successfully use the faster commands, output and tristate control bits must not be overwritten in the Capture-DR state of the JTAG TAP controller state machine. This behavior is undefined in the IEEE spec, and many chips do in fact overwrite the bits with constant values.

It is important to realize that by default, BSFlash will try to use the faster access method if the scan chain length is 255 or less. If the chip you are using does not recirculate bits correctly, this will cause failures when programming. You can force the program not to use these commands by inserting the following line into the ScanChain section of the description file:–

```
[ScanChain]
Access = Slow
```

5.8 Testing the scan chain

If you are unsure which bits get recirculated in the device you are using, you can use a special utility (ChainTest) that will analyze the EXTEST boundary scan chain of the chip. Use the following command line:–

```
ChainTest <chainpos> <IRlen> <DRlen> [<server>]
```

- **Chainpos** – Specifies the TAP controller to use. These are numbered from zero (counting from the left in the Multi-ICE server window).
- **IRlen** – This specifies the number of bits in the instruction register for this device.
- **DRlen** – This specifies the number of bits in the EXTEST boundary scan data register.
- **Server** – Specifies the host name of the computer running the Multi-ICE server. This parameter is optional – if absent it will be assumed that the Server is running on the same machine as the program.

The report from the program will usually span several pages, so you may want to redirect it to a file. Use the standard console redirection to achieve this. For example:–

```
ChainTest 0 5 214 >out.txt
```

The report from the program will look something like this:–

```
Boundary Scan Tester
Version 1.6

Scanning in EXTEST instruction...
Analysing scan chain...please wait...
Analysis complete...

Bit0 - 0,126,0,126 - Always captures 0
Bit1 - 126,0,0,126 – Recirculated
Bit2 - 0,126,0,126 - Always captures 0
Bit3 - 0,126,0,126 - Always captures 0
Bit4 - 126,0,0,126 – Recirculated
Bit5 - 126,0,63,63 – Indeterminate
Bit6 - 0,126,0,126 - Always captures 0
Bit7 - 126,0,0,126 – Recirculated
Bit8 - 0,126,0,126 - Always captures 0
Bit9 - 126,0,126,0 - Always captures 1
Bit10 - 126,0,0,126 – Recirculated
...etc
```

By matching this report up with your boundary scan description or BSDL file, you can see which bits will be re-circulated and which bits always capture a set value. Note that you only need to consider non-input bits that are actually used during programming. To illustrate this point: Imagine ChainTest reports that the enable bit for the WE pin always captures the value one. This is not a problem, because the WE pin is always enabled during programming (only the output value changes).

6 PROGCARDS

6.1 Description

The ‘ProgCards’ utility provides a scripting engine for all the programming methods that are supported by the programming suite. By reading descriptions from a set of ‘board files’, the program produces a menu of potential programming scripts that can be executed on the connected devices. By creating and combining board files, you can produce any desired programming system.

6.2 Syntax

```
ProgCards [<server>]
```

All configuration information is read from files in the current directory. The only command line argument to the utility is (optionally) the name of the machine on which the Multi-ICE server is running. This can be used to connect to a remote machine. If the argument is absent, the server is assumed to be running on the local machine.

6.3 Operation

The program will search the current directory for files with a .BRD extension. These files contain descriptions of the programming tasks can be performed, and may be altered by the user to achieve the desired functionality. The program will then connect to the Multi-ICE server and try to match the descriptions it has read with the devices connected to Multi-ICE. If several possible matches are made, a menu will be presented and the user must make a choice.

Each board file contains a description of a set of devices (using the names that appear in the Multi-ICE server window). ProgCards searches the entire scan chain (all of the devices shown in the Multi-ICE server window) starting with the first device (TAP #0) and looks for matches with the sets of devices described in the board files. If there is more than one match, a menu will be constructed and the user has to make a choice. ProgCards then executes the programming steps contained in the chosen board file and moves on to the next position in the scan chain, looking for new matches.

6.4 Writing a board file

Each board file contains the following information:-

- **A set of devices/TAPs**. These are described using the Multi-ICE driver names, in the order in which they occur in the JTAG scan chain. For each device position, you can specify several possible device names, or a wildcard that matches any device. Progcards 2.00 and above also supports the use of TAPs as an alternative to devices, this only changes the way the ‘devices’ are numbered. This is the preferred numbering method.
- **The programming steps** necessary to configure the matched devices. There can be any number of steps from zero up. Each step names a particular programming method, and specifies one of the matched devices. Depending on the particular method there may also be other parameters. Note that the devices are numbered from zero, starting with the first device that matches.
- **A priority**. If several matching configurations are found, the one with the highest priority will be used. A menu will only be presented if several configurations are matched with the same priority (and no higher-priority configuration is present). The priority scheme allows automatic programming setups to be created that will

perform the correct configuration automatically (without presenting a menu) even when potential matches overlap.

6.4.1 An example board file

```
[General]
Name = ARM7TDMI Header
Priority = 5

[ScanChain]
TAPs = 4
TAP0 = UNKNOWN,XC4062XLA
TAP1 = XC9572
TAP2 = XC9572
TAP3 = UNKNOWN,ARM7TDMI

[Program]
SequenceLength = 6

; download FLASH programmer
; to PLD
Step1Method = PLD
Step1TAP    = 2
Step1File   = AtmelFlashVia_CM_9572.svf

; and to FPGA
Step2Method = FPGA
Step2TAP    = 0
Step2File   = AtmelFlashVia_CM_4062.svf

; Program the FLASH chip
Step3Method = FLASH
Step3TAP    = 0
Step3File   = header_card_fpga.bit

; Verify that the programming worked
Step4Method = FLASHVERIFY
Step4TAP    = 0
Step4File   = header_card_fpga.bit

; Program the header-card PLD
Step5Method = PLD
Step5TAP    = 1
Step5File   = header_pld.svf

; Restore the serialiser
Step6Method = PLD
Step6TAP    = 2
Step6File   = bitstreamer.svf
```

The ‘device’ (old style) method of numbering devices is shown below.

```
[ScanChain]
Devices = 4
Device1 = UNKNOWN,XC4062XLA
Device2 = XC9572
Device3 = XC9572
Device4 = UNKNOWN,ARM7TDMI

[Program]
SequenceLength = 6

; download FLASH programmer to PLD
Step1Method = PLD
Step1Device = 2
Step1File   = AtmelFlashVia_CM_9572.svf
```

6.4.2 General comments

- The syntax is the same as that for a Windows .INI file. Anything after a semi-colon is a comment.
- The .BRD files may have any filename (so long as it ends with the correct extension). The text that is displayed in the menu, if one is generated, is taken from the ‘name’ field at the top of the file.
- If the ‘device’ method of numbering devices is used, then the device numbers start at one. However, the Device parameter for each programming step gives a zero-based offset from the first device matched. In the example above, `Step2Device = 0` refers to the first matched device. If the ‘TAP’ method of numbering is used then the TAP numbering starts at zero, and will avoid confusion.
- Filenames may be absolute or relative (to the current directory). The following examples are both valid:–

```
Step1File = C:\Work\bitfiles\peripheral.bit
Step3File = ../../supportfiles/header.bit
```

- Method names and step parameters are not case sensitive.
- In the device list, you may specify any number of possible device names for each position, separated by commas. Note that there should be no space between the commas and the device names. You may also specify a wildcard (a single ‘star’ character) which will match any device. For example:-

```
; match any device followed by a Virtex FPGA
[ScanChain]
TAPs = 2
TAP0 = *
TAP1 = XCV1000
```

- Text may be displayed at the end of a programming step. For example:-

```
Step1DisplayText = Version 2.0 now present
```

- A pause may be inserted at the end of a programming step, press a key to continue. For example:-

```
Step1Pause = 1
```

6.4.3 Supported programming methods

ProgCards supports a wide range of programming methods. These are described below.

6.4.3.1 FPGA download

This method downloads temporary configurations to Xilinx 4000-series FPGAs. The functionality is identical to the FPGA utility described in section 3. Here is an example step from a board file:-

```
Step2Method = FPGA
Step2TAP    = 0
Step2File   = viafpga.bit
```

The File parameter names a Xilinx bit file to download to the device. MCS format images are also supported, as for the FPGA utility (See section 3.2). See section 3.4.1 for notes about Xilinx 4000-series configuration.

6.4.3.2 Virtex / Virtex E download

This method downloads temporary configurations to Xilinx Virtex FPGAs. The functionality is identical to the FPGA utility described in section 3. Here is an example step from a board file:-

```
Step1Method = Virtex
Step1TAP    = 1
Step1File   = peripheral.bit
```

The File parameter names a Xilinx bit file to download to the device. See section 3.4.2 for notes about Virtex configuration.

6.4.3.3 PLD download

This method executes the contents of serial vector format files, used to program PLD devices. The functionality is identical to the PLD utility described in section 4. Here is an example step from a board file:–

```
Step5Method   = PLD
Step5TAP      = 2
Step5IRLength = 10
Step5File     = philipspld.svf
```

The IRLength parameter is optional. If present, it gives the length (in bits) of the JTAG instruction register for this device. If absent, ProgCards will assume an IR length of 8 (a Xilinx 9500 device). See section 4.4 for notes about PLD programming.

6.4.3.4 Boundary scan flash programming

A subset of the functionality of the BSFlash utility (described in section 5) is supported by ProgCards. There is no capability for reading data from a flash chip to a file and the test mode described in section 5.6 is not supported. Here is an example programming step:-

```
Step3Method   = BSFlash
Step3TAP      = 2
Step3DescFile = flash.fld
Step3File     = image.raw
Step3FileType = Binary
Step3Address  = 1C000
```

- **DescFile** – this parameter gives the name of the flash description file. See section 5.5 for more details.
- **File** – this parameter gives the path to the file that will be written into the flash.

- **FileType** – There are three possible file types: `XilinxBit`, `MCS` and `Binary`. Note that specifying `XilinxBit` or `MCS` actually has the same affect. The utility can automatically distinguish between the two file types and will accept either. The `Binary` file type will write any file into the flash, byte for byte.
- **Address** – This parameter is optional, and specifies the starting address for the image in the flash. This is given as a hexadecimal number, with no leading characters (FEA1 is a valid address, but 0x8C is not). If the parameter is absent, the start address defaults to zero.

Verify is also supported. The parameters are identical, except for the method name:–

```
Step4Method   = BSFlashVerify
Step4TAP      = 2
Step4DescFile = flash.fld
Step4File     = image.raw
Step4FileType = Binary
Step4Address  = 1C000
```

6.4.3.5 Indirect flash programming (for Integrator platforms)

Several methods are provided for programming flash devices indirectly. These methods have been developed specifically for ARM Integrator boards and utilize special logic designs implementing a JTAG programming interface to a connected flash chip. Typically the designs will be temporarily downloaded to the FPGA in an earlier programming step. In some cases, the programming interface may be distributed across two devices. In the example file in section 6.4.1, you can see that the flash interface is downloaded in two parts to an FPGA and a PLD before the Flash programming method is used.

The specific details and possible arrangements of the flash interfaces are not documented here. There are currently three types of interface, one for Atmel AT49 flash chips, and two for Intel devices. All interfaces support programming and verify, and are always accessed through an FPGA device.

Atmel devices are programmed one byte at a time, with an initial erase cycle for the whole chip. The file formats supported are Xilinx bit files and MCS files. There is no provision for specifying a start address – the address fields present in MCS files can be used to load images at addresses other than zero.

```
; Program an Atmel flash device
Step3Method = Flash
Step3TAP    = 0
Step3File   = image1.mcs

; Verify the contents of an Atmel flash device
Step4Method = FlashVerify
Step4TAP    = 0
Step4File   = image1.mcs
```

Intel devices are programmed in blocks, with an erase cycle at the beginning of each block. An address parameter is provided to allow several bit files to be placed in one flash chip. The syntax is the same as for the BSFlash address parameter, described in section 6.4.3.4.

```
; Program an Intel flash device
Step5Method  = IntelFlash
Step5TAP     = 2
Step5File    = upperimage.bit
Step5Address = 2E000

; Verify the contents of an Intel flash device
Step6Method  = IntelFlashVerify
Step6TAP     = 2
Step6File    = upperimage.bit
Step6Address = 2E000
```

To program Intel devices on Virtex based Integrator logic modules, the Virtex programming method is used for the first step to download a flash programmer into the FPGA. The next step(s) then use this design to talk to the flash device.

```
; Configure the FPGA with the flash programmer design
Step1Method = Virtex
Step1TAP    = 0
Step1File   = IntelFlashVia_LM_XCV1000.bit




; Program an Intel flash device
Step2Method = IntelFlash
Step2TAP    = 0
Step2File   = example.bit


; Verify the contents of an Intel flash device
Step3Method = IntelFlashVerify
Step3TAP    = 0
Step3File   = example.bit
```

To program Intel devices on Altera based Integrator logic modules, the flash programmer is pre-loaded into the PLD on the board, and it is therefore not necessary to download a temporary flash programmer design first. The IntelFlash and IntelFlashVerify methods are used as for Virtex.

```
; Program an Intel flash device
Step2Method = IntelFlash
Step2TAP    = 0
Step2File   = example.rbf


; Verify the contents of an Intel flash device
Step3Method = IntelFlashVerify
Step3TAP    = 0
Step3File   = example.rbf
```

Note: That the binary file must have the extension .rbf so that the file type can be recognized by progcards.

To program Intel devices on the Integrator AP motherboard, the FPGA programming method is used for the first step to download a flash programmer into the FPGA. The next step(s) then use this design to talk to the flash device.

```
; Configure the FPGA with the flash programmer design
Step1Method = FPGA
Step1TAP    = 2
Step1File   = IntelFlashVia_AP_XC4085XLA.bit

; Program an Intel flash device
Step2Method = APIntelFlash
Step2TAP    = 2
Step2File   = data.rbf
```

```
; Verify the contents of an Intel flash device
Step3Method = APIntelFlashVerify
Step3TAP    = 2
Step3File   = data.rbf
```

Note that default address for the AP system flash programming is 0x24000000.

6.4.4 Programming tips

- It is often useful to be able to skip a matched configuration (if, for example, you want to be able to program just one set of devices in a long scan chain). To achieve this, create a new board file that has the same device list as the real configuration. Set this to the same priority (so it is included in the menu) and set the SequenceLength field to 0. Change the name to something sensible (e.g. “ARM720T Header (Skip)”). When selected, this configuration will do nothing, and ProgCards will move on to the next set of devices in the chain.
- If there are devices in the scan chain which are not to be programmed, you can do something similar. For example, imagine a scan chain with two sets of devices to be programmed, separated by an additional device that should be ignored. Suppose that this device is recognized as “UNKNOWN” by Multi-ICE. Create a new board file with just this single device, and make the SequenceLength 0 so that no programming is performed. This will achieve the desired effect, but the configuration may also match the beginning of one of the two sets of devices (because FPGAs are also listed as “UNKNOWN”). If you do not want this to be reflected in the menu, make sure that your new board file has a lower priority than the others. ProgCards will notice this, and only include the higher-priority configurations.
- If you have set up several configurations for a particular board and want to make the program automatically choose a specific one, just increase the priority of that board file. If only one file has the highest priority, no menu will be generated (programming will be automatic). You do not have to delete the other configurations, and can easily return to the menu system by changing the priority back.