

ISP and STAPL

The ability to reprogram a device that has already been mounted onto a system board is referred to as In-System Programming (ISP). Although there are two types of ISP, microprocessor and external, this application note focuses entirely on microprocessor ISP – the technique used to configure Actel's ProASIC^{PLUS} devices. The microprocessor ISP method uses no outside programmer to configure the FPGA and requires programming control as well as the data for configuring the device. Programming control comes from the system itself, using the native processor, the operating system, and memory. The data could be resident in ROM (as in the case of predesigned configuration alternatives), arrive via some other media (such as a floppy disk), or transfer a communication channel like a network connection.

Details of the implementation vary according to the system's processor, operating system, bus structure, and memory architecture. This method of ISP works because the system processor has access to the IEEE 1149.1 (JTAG) port of the device. A software program then uses a data file to configure the FPGA.

Microprocessor ISP uses its own system for reconfiguration, and the details of the implementation vary according to the system's processor, operating system, bus structure, and memory architecture. This method of ISP works because the system processor has access to the IEEE 1149.1 (JTAG) port of the device. A software program then uses a data file to configure the FPGA.

The data file is generated by a program called the Standard Test and Programming Language (STAPL) Composer. The Actel Composer uses information about the programming of Actel ProASIC^{PLUS} devices as well as the JTAG scan chain information for a single device to generate a STAPL file for that device. The program that interprets the contents of the STAPL file is called a STAPL Player. The data format used for programming ProASIC^{PLUS} devices is a JEDEC standard known as the STAPL format. (The JEDEC STAPL standard, JESD71, can be obtained at: www.jedec.org).

The STAPL Player reads the STAPL file and executes the file's programming instructions. Because all programming details are in the STAPL file, the STAPL Player itself is completely device-independent. In other words, the system does not need to implement any programming algorithm details; the STAPL file provides all of the details.

Because ProASIC^{PLUS} devices are programmed through a JTAG port, it is possible to program multiple devices on a JTAG chain. Some considerations for chain programming are addressed in the "Chain Programming" section on page 7.

The operation of the STAPL Composer is of no interest to customers once they have received a STAPL file, so this application note concentrates on the software requirements for the STAPL Player and the STAPL file. The hardware issues for ProASIC and ProASIC^{PLUS} devices are described in more detail in Actel's application note, *Performing Internal In-System Programming Using Actel's ProASIC^{PLUS} Devices*.

Figure 1 on page 2 illustrates the procedure.

STAPL Player

Overview

The STAPL Player consists of two basic portions:

- High-level generic source code that executes the main program
- Low-level system-specific routines

STAPL Players can be implemented in either interpreted or compiled form. Altera supplies an interpreted version of the STAPL Player, which consists of high-level C code and low-level API routines. The code from Altera is supplied in the public domain. The Actel STAPL Player modifies the Altera code to work with Actel's FlashPro. Code for the Actel STAPL Player is on Actel's web site at <http://www.actel.com/products/proasicplus/info.html>

Note that executing STAPL statements based on the IEEE 1149.1 interface requires information on the location of the target device(s) along the serial chain. The method for conveying this chain information is platform-dependent - it may be specified via a user interface, in the STAPL file using the POSTDR, POSTIR, PREDR, and PREIR instructions, or read from a chain file (see the "Chain Programming" section on page 7). FlashPro uses a fourth method – automatic detection of all devices in the chain.

In addition to processing the STAPL file, the STAPL Player has the following functions:

- Check the CRC of a STAPL file (without executing the STAPL file)
- Access to the signals of an IEEE 1149.1 interface

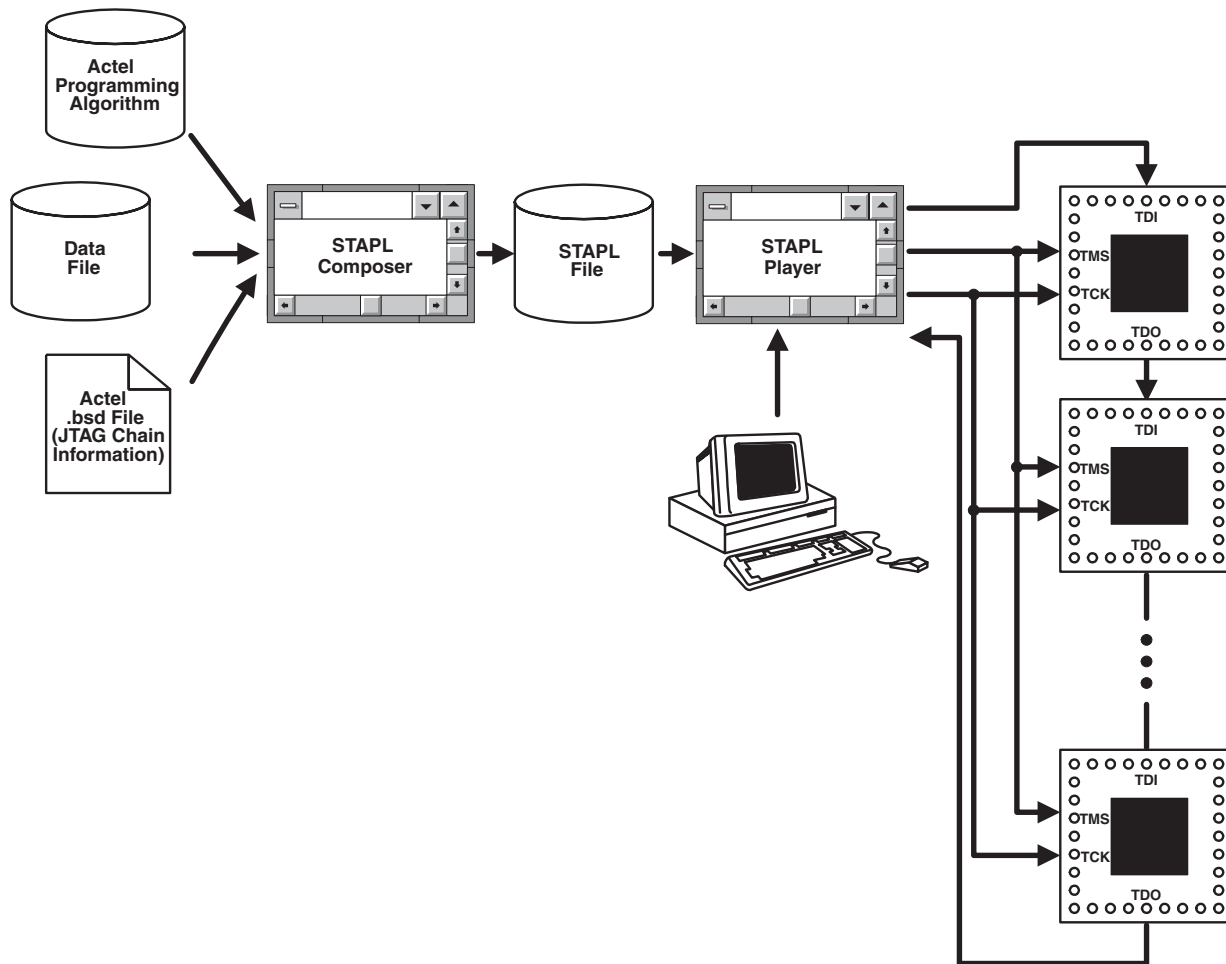


Figure 1 • Microprocessor ISP of Actel ProASIC^{PLUS} Devices

- Access to the signals specified by VECTOR/VMAP instructions as supported by the platform (this feature is not used in Actel STAPL files)
- Reliable mechanism for creating accurate real-time delays
- Reports exit status information following the execution of a STAPL file (e.g., an exit code)

Command-Line Interface

Actel's STAPL Player can be run in command-line mode in a DOS or Windows system. The command has the form

```
COMMAND? [optional switches]
```

The options are summarized as follows:

- a <action> – Specify a specific action (programming, verification, etc.) as defined in the STAPL file.
- d <variable=value> – Initialize a variable
- h – Show help messages
- v – Verbose option (show all messages)

The STAPL Player API

The Application Specific Interface (API) contains a series of low-level functions that interact with the processor used for system control as well as its operating system. The API function set used with Actel's STAPL Player is shown below. Note that the exact low-level code changes depending on the processor. The process consists of modifying one of the generic C-source files in the STAPL Player (jamjtag.c in Version 2.2) and adding device-specific files.

The key file in the generic STAPL Player collection is the file jamjtag.c. This file can be edited to replace the generic, device-independent routines with device-specific ones. If a microprocessor I/O port is used to drive the JTAG port, only one generic JTAG function, jam_jtag_io, is needed.

On the other hand, a more-complicated sequence is required if a custom JTAG controller is used. A short example of the jam_jtag_reset_idle sequence is given below. The intent of the routine is to perform a Test Logic Reset followed by stepping the state machine to the Run Test/Idle state. Note that the definition of a variable, in this

case HW_SPECIFIC, causes the device-specific routines JtagReset and JtagIdle to be called.

```

/*****
*****/

/*      */

void jam_jtag_reset_idle(void)

/*      */

/*****
*****/

{
#ifdef HW_SPECIFIC

    JtagReset();
    JtagIdle();

#else

    int i = 0;

    /*
     * Go to Test Logic Reset (no matter what the
     * starting state may be)
     */
    for (i = 0; i < 5; ++i)
    {
        jam_jtag_io(TMS_HIGH, TDI_LOW,
        IGNORE_TDO);
    }

    /*
     * Now step to Run Test / Idle
     */
    jam_jtag_io(TMS_LOW, TDI_LOW, IGNORE_TDO);

#endif

    jam_jtag_state = IDLE;
}

```

There are two categories of functions. One set, listed in Table 1 includes those that manipulate the JTAG port (these routines are specific to Actel's FlashPro), while the other set provides operating system services (Table 2).

Table 1 • JTAG API Functions

Function	Description
JtagReset	Send TMS=1,1,1,1,1 to go to Test Logic Reset State, or send TRST=1 if TRST is implemented
JtagIdle	Traverse the state machine from Pause to Idle
JtagIrEnter	Traverse the state machine from Idle or Reset to Shift-IR or Pause-IR
JtagDrEnter	Traverse the state machine from Idle or Reset to Shift-DR or Pause-DR
JtageShift	Shift N data bits starting at bit 0 of data[0].
JtagRead	Return data output on TDO from previous shift sequence
JtagWait	Pause until M TCK cycles have been output

Table 2 • OS API Functions for FlashPro

Function	Description
stp_getc	Returns a single character from the STAPL file
stp_seek	Moves the pointer within the STAPL file
stp_jtag_io	Low-level JTAG I/O function
stp_message	Returns a message to the STAPL Player
stp_delay	Generates a delay loop for n seconds
stp_malloc	Allocates memory
stp_free	Frees up memory

After the API has been written, it is compiled and linked with the C-based STAPL Player, and the final executable can then be run in the system. The next two sections describe the general STAPL file format and the specific details of an Actel STAPL file.

STAPL File Overview

The STAPL Player operates on a STAPL file which consists of a sequence of the following program elements:

- NOTE statements are comments indicating the contents and features of the file. The key strings, or NOTE contents, are given in quotation marks.
- ACTION statements describe the sequences of steps required to implement a complete operation. An example is PROGRAM for programming a device.
- PROCEDURE blocks contain STAPL statements describing computations as well as interactions with JTAG-compliant devices.
- DATA blocks declare variables and their values (variables MUST be declared).
- A single CRC statement contains the cyclic redundancy code that verifies the data integrity of the file.

Note that there must be at least one ACTION statement and at least one PROCEDURE or DATA block. Variables declared inside a PROCEDURE block are only available inside that block. Variables declared inside a DATA block are available in and shared by any PROCEDURE block that uses that DATA block. The PROCEDURE and DATA blocks end with ENDPROC and ENDDATA, respectively. STAPL files have variables of two types: INTEGER (32-bit signed) and BOOLEAN (SIMILAR to single-bit unsigned). One-dimensional Boolean or integer arrays are also supported, but multidimensional arrays are not. String variables are NOT supported other than a simple messaging facility.

Arithmetic, logical, and relational operators are provided for integers as well as logical operators for Boolean expressions. However, there are no array operators for either integer or Boolean arrays. Case sensitivity is not required (with one exception not discussed in this application note). Like the JEDEC spec, this document will use upper-case instruction and keyword names and lower-case label and variable names, but this is NOT required in the language.

Statements and Keywords

Each statement in a STAPL file contains up to three elements: a label (optional), an instruction, and arguments. The number and type of arguments depends on the instruction. A semicolon (;) terminates the statement. Labels, as in many programming languages, provide a method of branching.

ACTION	EXIT	NEXT	PUSH
BOOLEAN	EXPORT	NOTE	STATE
CALL	FOR	POP	TRST
CRC	FREQUENCY	POSTDR	WAIT
DATA	GOTO	POSTIR	VECTOR
DRSCAN	IF	PREDR	VMAP
DRSTOP	INTEGER	PREIR	
ENDDATA	IRSCAN	PRINT	
ENDPROC	IRSTOP	PROCEDURE	

In addition, the sixteen state names shown in the IEEE 1149.1 state diagram in Figure 2 are also reserved.

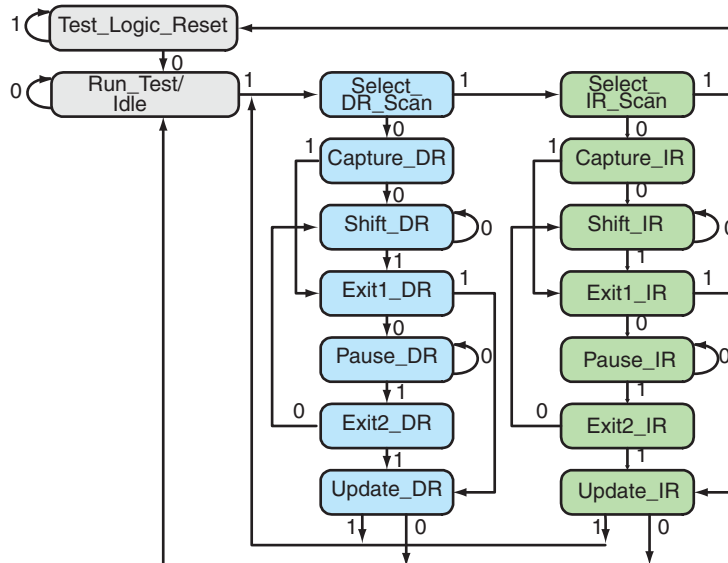


Figure 2 • State Diagram (Showing Reserve STAPL State Names)

A small number of other strings are also reserved; several of these will be discussed in the command explanations that follow.

BOOL	CYCLES	RECOMMENDED	USEC
CAPTURE	INT	STEP	USES
CHR\$	MAX	THEN	
COMPARE	OPTIONAL	TO	

Program Flow and Important Commands

A STAPL file, such as Actel's example in "Appendix A" on page 10, may include multiple ACTION statements. When the STAPL Player executes a STAPL file, only one of those actions will be performed. Execution continues with calls to the list of PROCEDURE blocks listed in the ACTION statement. Each PROCEDURE block name may be followed by one of the keywords RECOMMENDED or OPTIONAL. The RECOMMENDED keyword indicates that the PROCEDURE block will be called unless explicitly declined by the user, while the OPTIONAL keyword indicates that the PROCEDURE block will not be called unless explicitly requested by the user.

Execution terminates either when the end of the ACTION statement is reached or when an EXIT statement is processed. The flow of execution within each of the called PROCEDURE blocks is controlled using three methods – branches, calls to other PROCEDURE blocks, and loops.

STAPL uses a stack scheme for managing subroutine calls and loops. Nested activities, included in the stack, are ACTIONS, CALL / ENDPROC, FOR / NEXT, and PUSH / POP. When an ACTION, CALL, FOR, or PUSH statement is encountered, information about the operation is added to the stack. When the corresponding ENDPROC, NEXT, or POP statement is encountered, the record is removed from the stack. (For the NEXT statement, the stack record is removed only when the loop has run to completion.)

A PROCEDURE block may call itself. To make a PROCEDURE block re-entrant, the PUSH and POP statements may be used to save and restore the values of data variables on the stack. The PUSH statement is used to save data variables before the call, and the POP statement is used to restore values after the return. In that case, the PROCEDURE block may call itself to implement recursion. If this form of recursion is employed, the resulting stack depth must be determinate.

The branch is executed with a GOTO statement that causes execution to jump to the statement that corresponds to the label. This label must be located within the same PROCEDURE block as the GOTO statement and may or may not have been encountered already in that PROCEDURE

block. The IF statement can be used with the GOTO statement to create a conditional branch.

The CALL statement causes execution to jump to a PROCEDURE and the location of the CALL statement is saved on the stack. When execution of the called PROCEDURE block has reached completion by encountering an ENDPROC statement, execution jumps to the statement following the CALL statement, and the record is deleted from the stack.

The IF statement can be used with the CALL statements to call a subroutine conditionally.

The FOR statement is used for iteration or "looping." Each FOR statement has an "iterator," an associated integer variable that maintains the iteration count. When a NEXT statement using the same iterator variable is encountered, the iterator is compared to its terminal value. If the iterator has reached its terminal value and the body of the loop has been executed for the last time, the FOR loop is complete and control is passed to the statement following the NEXT statement. Otherwise, the iterator is incremented (or stepped, if the STEP keyword is used with the FOR statement) and control jumps back to the statement following the FOR statement. FOR statements can be nested.

The IF/THEN construct, as in many programming languages, allows conditional operations. The IF statement evaluates a Boolean expression, and if the expression is true, executes a statement. The THEN statement can be any statement type except ACTION, BOOLEAN, CRC, DATA, ENDDATA, ENDPROC, INTEGER, NOTE, and PROCEDURE statements). All IF statements must be located within PROCEDURE blocks.

The DRSCAN statement specifies an IEEE 1149.1 data register scan pattern to be loaded into the target data register. The scan data shifted out of the target data register may be captured in a Boolean array variable, compared to a Boolean array expression, a combination of the two, or it may be ignored. The data register length is a nonzero integer expression specifying the number of data bits to be shifted. The scan data array is a Boolean array expression, specifying the data to be loaded into the data register. The data is shifted in increasing order of the array index, i.e., beginning with the lowest index. The capture array is a Boolean array variable. The compare array and mask array are Boolean array expressions, and the result, is a Boolean variable or a single element of a Boolean array variable that receives the result of the comparison. Mask array bit values of "1" represent bits to be compared, and bit values of "0" represent bits not to be compared. A successful comparison will cause a one (or TRUE) value to be stored in the result variable. An unsuccessful comparison will cause a zero (or FALSE) value to be stored in the result variable but will not

interrupt the STAPL file execution. To abort in the case of an error, a conditional (IF) statement must be used to test the result value, and the EXIT statement called to stop the program. All DRSCAN statements must be located within PROCEDURE blocks. The DRSTOP statement, if used in conjunction with the DRSCAN statement, specifies the IEEE 1149.1 end state for data register scan operations. This end state must be one of the IEEE 1149.1 states: RESET, IDLE, IRPAUSE, or DRPAUSE. The default state is IDLE. An example of the use of the DRSCAN and DRSTOP statements is given in the sample file in “Appendix A” on page 10.

The Instruction Register equivalents to DRSCAN and DRSTOP are IRSCAN and IRSTOP.

The STATE statement causes the IEEE 1149.1 state machine to go to the specified state. The path can specify any state, but the last state must be one of the following: RESET, IDLE, DRPAUSE, or IRPAUSE.

The TRST statement enables the optional IEEE 1149.1 TRST pin for the specified number of TCK clock cycles and/or for a minimum number of microseconds. A TRST statement may specify a clock cycle count, a time delay, or both. When both are specified, the clock cycles and time delay occur simultaneously until both are satisfied. When a USEC time delay is specified, the delay implemented is not related to the clock rate of TCK. TCK may continue to run during the USEC delay, or it may be stopped in the low state. Since many IEEE 1149.1 compliant devices do not offer this optional TRST pin, this statement will not ensure that all devices on the chain are reset to the TEST-LOGIC-RESET state. To ensure reset of all devices to this state, use the STATE RESET statement.

The WAIT statement causes the IEEE 1149.1 state machine to go to the specified wait state for the specified number of TCK clock cycles, and/or for a minimum number of microseconds. A WAIT statement must specify a clock cycle count, a time delay, or both. When both are specified, the clock cycles and time delay occur simultaneously until both are satisfied. When a USEC time delay is specified, the delay implemented is not related to the clock rate of TCK. TCK may continue to run during the USEC delay, or it may be stopped in the low state. A maximum number of TCK clock cycles and/or a maximum number of microseconds may also be specified. If either the wait state or the end state is not specified, IDLE is assumed. If an ENDSTATE is specified, the IEEE 1149.1 state machine will go to that state immediately after the specified number of clock cycles and the specified amount of real time has elapsed. The valid wait state and end states are IRPAUSE, DRPAUSE, RESET, and IDLE.

The FREQUENCY statement is an extension to the STAPL specification. It defines the maximum frequency at which the IEEE 1149.1 TCK signal should operate following execution of this statement. The specified frequency remains in effect until a subsequent FREQUENCY statement is executed or STAPL file execution terminates. Using an integer expression sets the frequency to a new value. Omitting this integer expression restores the operating frequency to the value operating prior to the execution of the first FREQUENCY statement encountered during the STAPL session. The player can choose to run at or below this frequency for the entire operation of the STAPL file, if no dynamic frequency change is possible. If the underlying hardware cannot dynamically change the clock frequency and cannot support a frequency less than or equal to this setting, an error occurs.

The execution evaluates to an integer representing a frequency in Hertz and must be greater than or equal to zero, although the result when the clock frequency is zero is undefined. If this statement is employed, an accompanying FREQUENCY_MIN note field is mandatory using the slowest value of all FREQUENCY statements in the STAPL file. The Actel STAPL files are set up to use TCK as a free-running clock source. TCK is intended to indicate exactly the frequency desired, although the specification allows any frequency below the requested frequency. Note that in the example of “Appendix A” on page 10, the 'freq' variable is set to 4 (later it is multiplied by 106 to give a TCK frequency of 4 MHz). Many users will probably use RCK as the clock source. This will require either modification of the STAPL file or. First, Boolean variable USE_RCK needs to be set to '1'. In addition, the 'freq' variable should be changed to reflect the actual RCK frequency in use. Both of these changes can be made via the command line using the -d switch described above for setting variable values.

The single CRC statement should always be the last statement in a STAPL file; any characters located after the CRC statement will not be included in the CRC computation. The CRC is a 16-bit convolution code based on a generator polynomial. CRCs for STAPL files are calculated using the generator polynomial employed by the CCITT for 16-bit CRCs:

$$G(X) = X^{16} + X^{12} + X^5 + 1$$

The CRC is calculated on all characters in the file, including comments and white-space characters but excluding carriage-return (CR) characters. The value obtained is then compared to the value found in the CRC statement. If the CRC values agree, the data integrity of the STAPL file is verified. A CRC value in the code font file of "0" indicates that CRC should not be compared.

Handling Errors

There are two kinds of errors that must be addressed. The STAPL file itself creates the first type of error. After the STAPL file has been executed, it returns an exit code that indicates the result of the programming. Actel's STAPL Player supports the exit codes shown in [Table 3](#).

Table 3 • STAPL File Exit Codes

Exit code	Description
0	Success
5	Entering ISP failure
6	Unrecognized device ID
7	Unsupported device version
8	Erase failure
11	Verify failure
12	Read failure
90	Unexpected RCK detected
91	Calibration data parity error

Unlike the first error type, the STAPL Player finds the second error type. For example, if the STAPL Player reads a STAPL file and finds a syntax error, the Player will generate an exit code indicating an error. [Table 4](#) lists the error codes that the STAPL Player can return.

Table 4 • STAPL Player Exit Codes

Exit code	Description
0	Success
1	Out of memory
2	I/O error
3	Syntax error
4	Unexpected end
5	Undefined symbol
6	Redefined symbol
7	Integer overflow
8	Divide by zero
9	CRC error
10	Internal error
11	Bounds error
12	Type mismatch
13	Assign to const
14	Next unexpected
15	Pop unexpected
16	Return unexpected
17	Illegal symbol
18	Vector map failed
19	User abort
20	Stack overflow
21	Illegal opcode
22	Phase error
23	Scope error
24	Action not found

Chain Programming

Chain programming is defined as programming several devices that are on the same JTAG chain. Two primary device configurations for this chain are as follows:

- ProASIC^{PLUS} devices ONLY, programmed one at a time
- ProASIC^{PLUS} devices ONLY, programmed concurrently
- ProASIC^{PLUS} devices and other JTAG devices, either programmable or nonprogrammable

Actel's STAPL file currently supports the first type of chain programming. When a file is used to store this chain information, the STAPL Player must support chain files as specified in the EIA/JEDEC Standard JESD32, "Standard for Chain Description File." If a chain has multiple programmable devices, they can successfully be programmed one at a time by bypassing the devices not being programmed. Concurrent programming will be supported in a future release.

In the case where nonprogrammable devices are in the chain, they must be bypassed and then the programmable device(s) can be programmed. If there are multiple ProASIC^{PLUS} devices, then they should be programmed individually.

It is not possible to concurrently program devices from different vendors. If non-ProASIC^{PLUS} programmable devices coexist on the same chain, then the non-ProASIC^{PLUS} devices should be treated just like nonprogrammable devices when programming the ProASIC^{PLUS} devices.

All JTAG-compliant devices have a bypass mode that bypasses the data register. Any device in bypass mode will have a data register length of 1. If an entire chain is in bypass mode, then the apparent length of the entire chain's data register is equal to the number of devices in the chain. When programming a single device, bypass all other devices. In this case, the complete length of the chain data register will be the length of the programmable device's data register plus one bit (corresponding to bypass mode) for other devices in the chain. This means that the bitstream intended for the data register of the programmable device must be padded on the front and back end by as many bits as there are devices before and after the programmable device ([Figure 3 on page 8](#)).

A device can be placed into bypass mode by loading an instruction of all 1s. This will vary from device to device only because different devices have different instruction registers lengths. Therefore, it is important to know the length of the instruction register for each device in the JTAG chain. [Figure 4 on page 8](#) illustrates this concept.

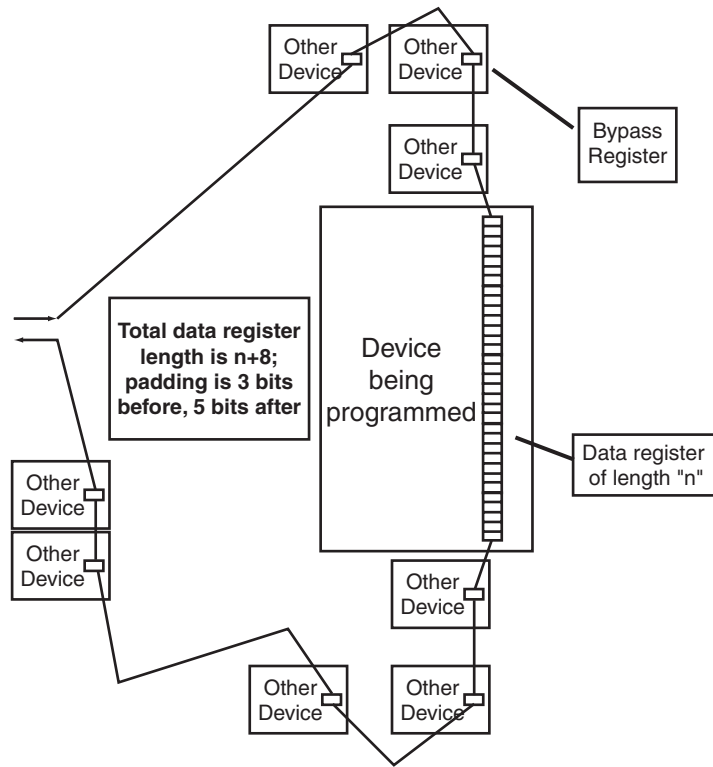


Figure 3 • Data Register Padding

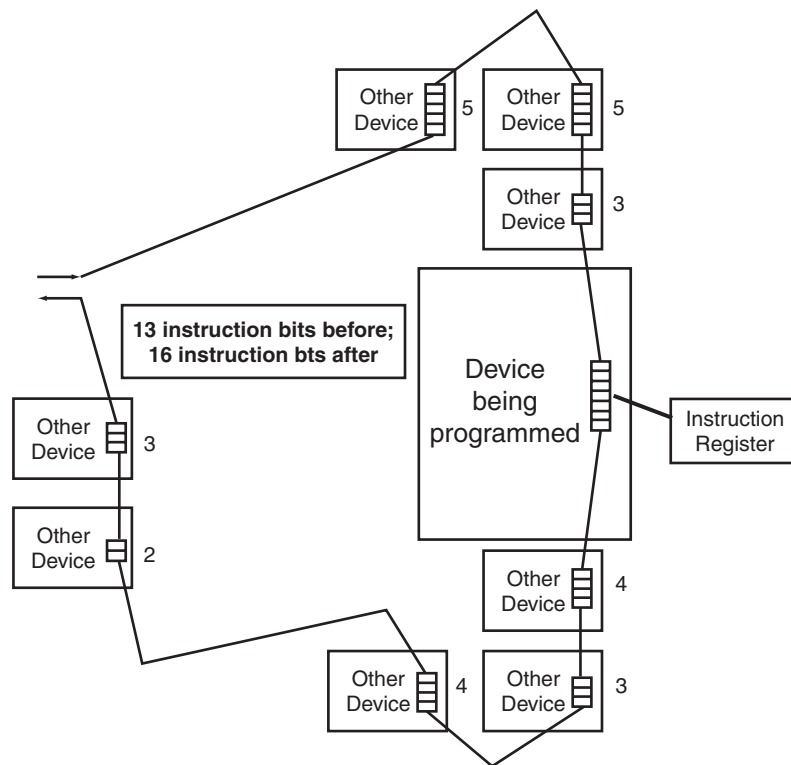


Figure 4 • Putting Devices into Bypass Mode

In order to account for more than one device on a chain, there are some instructions in the STAPL language that allow for the necessary padding of the bitstreams. The instructions can be added to the STAPL file, as described in Table 5. Note when configuring a bitstream, the first bits end up in the last device of the chain, so the padding at the front end of the bitstream is for the devices at the end of the chain.

These instructions must be added to the PROCEDURE INITIALIZE section of the STAPL file.

Table 5 • Register-Padding Instructions

Instruction	Purpose
PREIR n	Used to put the devices after the device being programmed into Bypass mode; n is the number of instruction bits after the device being programmed.
POSTIR n	Used to put the devices before the device being programmed into Bypass mode; n is the number of instruction bits before the programmable device
PREDR n	Used to pad the data by the number of bypassed devices after the device being programmed; n is the number of devices after the device being programmed.
POSTDR n	Used to pad the data by the number of bypassed devices before the device being programmed; n is the number of devices before the device being programmed.

Related Documents

Application Notes

Performing Internal In-System Programming Using Actel's ProASIC^{PLUS} Devices

http://www.actel.com/documents/APA_Microprocessor_AN.pdf

Appendix A

Sample STAPL File (Abridged)

```
NOTE "CREATOR" "map2bitstream 6b2.36";
```

```
NOTE "DEVICE" "APA750";
```

```
NOTE "DATE" "2002/05/06";
```

```
NOTE "STAPL_VERSION" "JESD71";
```

```
NOTE "IDCODE" "014101CF";
```

```
NOTE "DESIGN" "C:\Actelprj\apa_r2Test\designer\toptest";
```

```
NOTE "CHECKSUM" "B4F4";
```

```
NOTE "SAVE_DATA" "BITSTREAM";
```

```
NOTE "AMHOME" "C:\Des_r2_2001/am";
```

```
NOTE "ALG_VERSION" "6";
```

```
NOTE "MAX_FREQ" "10000000";
```

Notes

Recommended Indicates Called Procedure

```
ACTION PROGRAM = FIX_INT_ARRAYS RECOMMENDED, INITIALIZE,  
                DO_ERASE,  
                CHECK_SECURITY RECOMMENDED,  
                DO_PROGRAM,  
                DO_VERIFY_BOL RECOMMENDED,  
                PROGRAM_UROW,  
                POWER_DOWN;
```

Action Statements

```
ACTION ERASE = FIX_INT_ARRAYS RECOMMENDED, INITIALIZE,  
              DO_ERASE, POWER_DOWN;
```

```
ACTION READ_IDCODE = DO_READ_IDCODE;
```

```
ACTION VERIFY = FIX_INT_ARRAYS RECOMMENDED, INITIALIZE,  
              DO_VERIFY_EOL, POWER_DOWN;
```

```
ACTION QUERY_SECURITY = FIX_INT_ARRAYS RECOMMENDED, INITIALIZE, DO_QUERY_SECURITY, POWER_DOWN;
```

```
DATA PARAMETERS;  
BOOLEAN ULOP=1;  
INTEGER freq = 4; ' TCK frequency 4 MHz  
BOOLEAN USE_RCK=0;  
ENDDATA;
```

Named Data Blocks

```
DATA GV;  
BOOLEAN ID[32];  
BOOLEAN z[195];  
    o  
    o  
    o
```

```
INTEGER hex[16]= 70,69,68,67,66,65,57,56,55,54,53,52,51,50,49,48;
ENDDATA;
```

```
DATA DEVICE;
INTEGER tileSize[3]=12,32,32;
    ○
    ○
    ○
ENDDATA;
```

Boolean Data in Compressed Form



```
DATA BITSTREAM;
BOOLEAN DESIGN[84]= $B9AB970D997A638371D43;
BOOLEAN sw[763895]= @@JN00ey@t@@@l@@V@@@L2y10000008P0Lv@rEO000uM1mgggKjHgILj
HILLjXKLLDZKLLQZKlfQZKfgQZagwDL63mRggILL_@h7ux@@VO2y7004h1OZKv7t1@@@10bp00ho@Nq1@
    ○
    ○
    ○
t27Rqa6jk5EN9dXUEFdR_Tmx@Fmp@70p@@@V3yj@x@@zwzaTGv@VKiBhk@mslAmdJ00;
INTEGER CHECKSUM=46324;
ENDDATA;
```

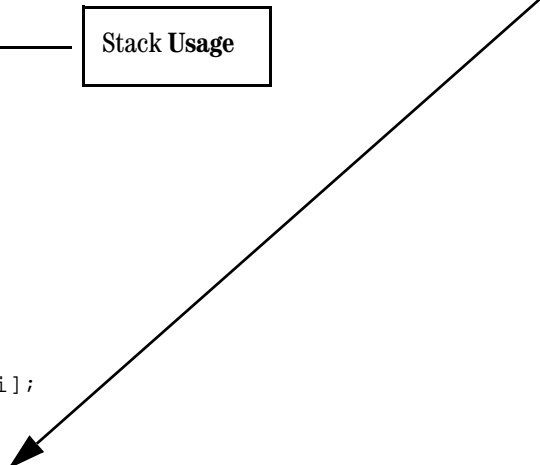
' JESD71 (Aug 99) specification is ambiguous about ordering of integer
' array initializers.
' Until the specification is clarified, we will allow both orderings

```
PROCEDURE FIX_INT_ARRAYS USES GV,DEVICE;
INTEGER ocheck[2] = 1,0;
IF ocheck[0]==0 THEN GOTO intok;
FOR i=0 TO 7;
    PUSH hex[i];
    hex[i]=hex[15-i];
    POP hex[15-i];
NEXT i;
PUSH tileSize[0];
tileSize[0]=tileSize[2];
POP tileSize[2];
FOR i=0 TO 87;
    PUSH tileType[i];
    tileType[i]=tileType[176-i];
    POP tileType[176-i];
NEXT i;
```

Stack Usage



Procedure Blocks



```
intok:
ENDPROC;

PROCEDURE INITIALIZE USES GV,PARAMETERS,GP,POWER_DOWN;
IF !USE_RCK THEN FREQUENCY freq*1000000 ;
WAIT RESET, 5 CYCLES;
IRSCAN 8,$0f;
DRSCAN 32,$000000FF,COMPARE $014101cf,$0BFFFFFF,PASS;
IF PASS==1 THEN GOTO idok;
STATUS=6;
CALL POWER_DOWN;
idok:
IRSCAN 8,$09;
IRSCAN 8,$0a;
IRSCAN 8,$92;
IF USE_RCK THEN DRSCAN 8,BOOL(freq-1);
IF !USE_RCK THEN DRSCAN 8,BOOL(128+freq-1);
CALL GP;
IRSCAN 8,$d3;
IRSCAN 8,$d8;
IRSCAN 8,$e5;
IRSCAN 8,$c8;
DRSCAN 1,#0,CAPTURE PS[];
ENDPROC;
```

Loading Instruction Register

Loading Data Register

CRC 8772;

Actel and the Actel logo are registered trademarks of Actel Corporation.
All other trademarks are the property of their owners.



<http://www.actel.com>

Actel Corporation

955 East Arques Avenue
Sunnyvale, California 94086
USA

Tel: (408) 739-1010

Fax: (408) 739-1540

Actel Europe Ltd.

Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom

Tel: +44 (0)1276 401450

Fax: +44 (0)1276 401490

Actel Japan

EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan

Tel: +81 03-3445-7671

Fax: +81 03-3445-7668

Actel Hong Kong

39th Floor
One Pacific Place
88 Queensway
Admiralty, Hong Kong
Tel: 852-22735712