ISR Programming Using an Embedded Processor with Jam™

Introduction

In System Reprogrammability™ (ISR™) allows programmable logic devices to be reprogrammed after being soldered onto a printed circuit board. This capability simplifies device prototyping, the manufacturing cycle, and field upgrades.

One method for programming ISR devices is to utilize an embedded processor that already exists on the target system. The embedded processor, executing ISR instructions, reads data from a data source and programs a chain of ISR devices.

The Jam™ Programming and Test Language, a freely licensed open standard, is designed to program any ISR programmable devices that use the IEEE standard 1149.1 JTAG (Joint Test Action Group) interface.

This application note illustrates how to use the Jam language to program ISR programmable logic devices using an embedded processor. It includes a detailed examination of the Jam File, the Jam Player, and system requirements such as memory and the interface between the ISR devices and the embedded processor.

The Jam Language

The Jam language is an interpreted language, meaning the Jam program source code is executed directly by an interpreter program (such as the Cypress Jam Player) running on a host computer, without first being compiled into binary executable code. As the interpreter executes the Jam program, signals generated on the IEEE 1149.1 JTAG interface will program the ISR devices.

The Jam language consists of three major parts—the Jam Composer, the Jam Player, and the Jam File. Cypress's Jam Composer software is capable of generating a Jam File (.jam). The Jam File contains the programming algorithm and the data for a chain of JTAG-compliant ISR devices. The Jam Player, running on the embedded processor, reads, interprets, and executes a Jam File, sending a binary bit stream through a JTAG interface to the ISR devices. When reprogramming the ISR devices, only the Jam File changes. The Jam Player source code remains the same. *Figure 1* shows a block diagram of an embedded processor using Jam to program devices.

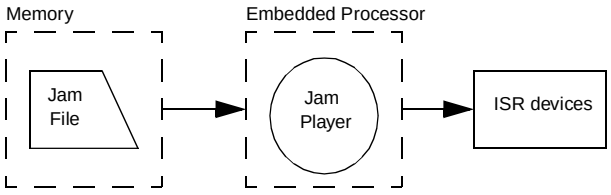


Figure 1. Block Diagram of Embedded Processor ISR using a Jam File and Jam Player

System Considerations

A simple embedded system consists of an embedded processor, EPROM or system memory, the ISR devices, and some interface logic. While programming the ISR devices, the embedded processor transfers data from the system memory or EPROM to the ISR devices. *Figure 2* shows a block diagram of an embedded system. This example has four ISR devices connected in the JTAG chain. The JTAG chain can connect directly to four of the data pins on the embedded processor. Using the interface logic saves these four ports by treating the JTAG chain as an address location on an existing bus. Furthermore, installing a 10-pin ISR programming cable header on the printed circuit board allows the Cypress Jam Player and the Cypress ISR programming cables to access the ISR devices in the JTAG chain for programming or verifying.

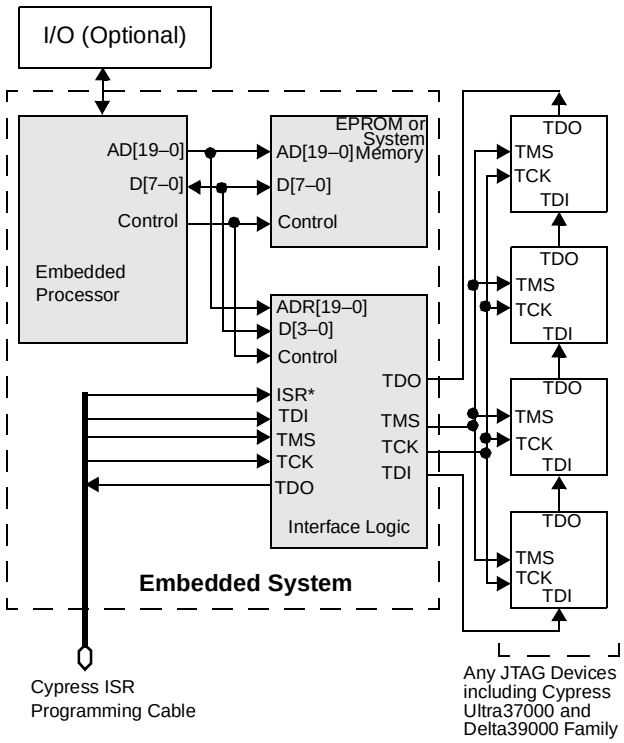


Figure 2. Block Diagram of Embedded System

Interface Logic

An example of the interface logic is shown in *Figure 3*. When the interface logic receives the proper address and control signals, it synchronizes the timing of the TDI, TCK, and TMS signals and drives the output pins via a multiplexer. The multiplexer also allows the Cypress ISR programming cables to access the JTAG chain for programming or verifying. The timing of the TDI, TCK, and TMS signals are synchronized using

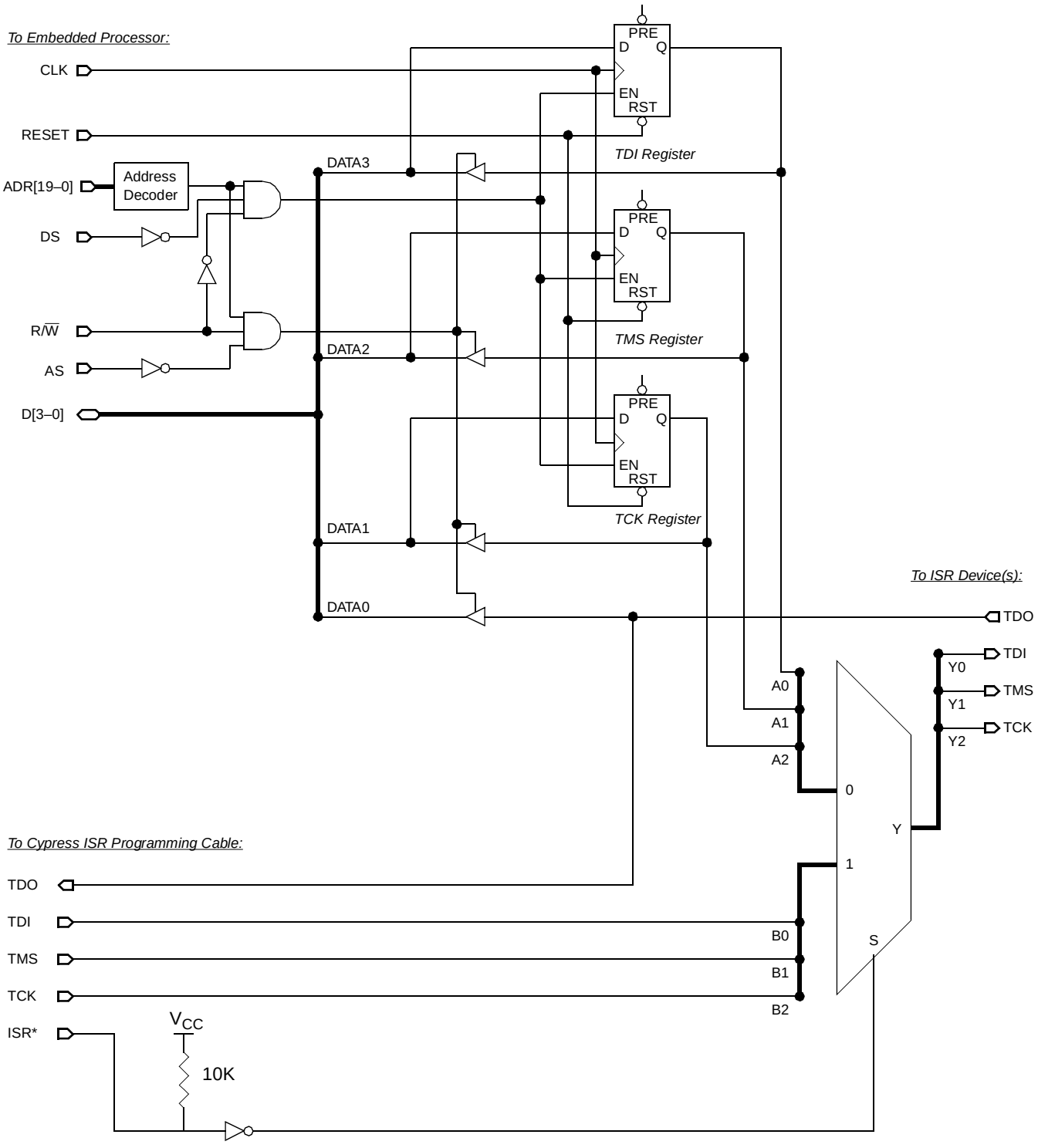


Figure 3. Interface Logic

the registers and the CLK signal from the embedded processor. The buffer on TDO prevents bus contention. The buffers on TDI, TCK, and TMS can be included to read back the values in the registers for debug purposes. The AND gates control whether a read or write operation is performed using the R/$\overline{W}$ signal. The AS and DS signals add another level of select/deselect to the circuitry.

Jam File Structure

A Jam File contains the following: a Note Field Section, a Variable Declaration and Initialization Section, and an Algorithm Section, as shown in *Figure 4*.

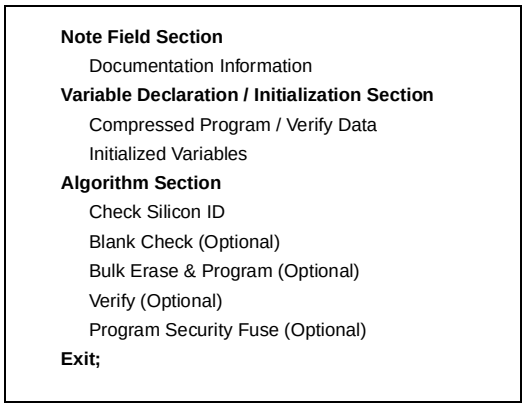


Figure 4. Jam File Structure

The Note Field stores information about the Jam File, such as the name of the device, the creation date of the Jam File, the version of the Jam language specification used, etc. The Jam Player can extract this information from the Jam File without actually executing the Jam File.

The Variable Declaration and Initialization Section consists of the program/verify data and other declared variables used in the Jam File. The initialization of other types of variables is optional. If the variable is a Boolean array, it can be initialized as a compressed data array—ACA (Advanced Compression Algorithm). For more detailed information on the ACA data format, refer to the *Jam Programming & Test Language Specification.*

The Algorithm Section contains the commands and code needed to program the target device. It contains all of the functions that can be performed on the target devices, such as blank check, erase, program, verify, etc., as well as any branching or looping that is required.

Generating a Jam File

The Jam File is an ASCII file generated by the Cypress Jam Composer software. The Jam File can then be stored in the EPROM, FLASH, or system memory of the embedded system, after converting it to an Intel-formatted Hexadecimal File (.hex) or similar programming file. Some embedded processor software packages or other utilities can automatically convert the Jam Files. Most EPROM programmers support *raw binary* or *absolute binary* formats allowing the programmer to read the Jam File without conversion.

The Jam Player

The Jam Player is a C program that resides permanently in program storage, such as ROM. It parses the Jam File, interprets each instruction, and reads and writes data to and from the JTAG port. Cypress provides the source code for the Jam Player. The main program of the Jam Player performs all of the basic functions, such as parsing and interpreting, and does not need modifying. The I/O functions of the Jam Player, such as I/O pin addressing, routines for delays or file I/O, and operating system-specific functions, are contained in the file jamstub.c. The Jam Player can be adapted to any application or system architecture by modifying this file. Any change or upgrade to the ISR devices requires a change to the Jam File only.

Jam Player Operations

The Cypress Jam Player can perform the following operations:

- Execute a Jam File.
- Check the cyclic redundancy code (CRC) of a Jam program (without executing the program).
- Extract information from the NOTE fields (without executing the program).

These operations share the basic functions that process Jam language statements. Thus, they are combined in a single software module and implemented as three separate exported functions in the Jam Player software interface. Each of these operations are explained below.

Executing a Jam File

The Jam Player executes a Jam File by reading and executing each program statement in the Jam File. The Jam Player processes all statements in the Jam File except for NOTE statements and CRC statements. The Jam Player ignores these statements while executing the Jam File. The DRSCAN, IRSCAN, WAIT, and STATE instructions in the Jam File cause the Jam Player to interact with the IEEE 1149.1 JTAG interface. The Jam Player will terminate when it encounters an EXIT statement.

Information can be passed into the Jam Player at the time of execution. Some of this information is required, while some of it is optional.

The following are Jam Player input sources:

- The Jam File.
- Information that can be obtained from the IEEE 1149.1 JTAG hardware interface during the execution of a Jam File.

The Jam Player can be run with the following flags:

```
jam [-h] [-v] [-m<bytes>] <Jam File>
```

Flags in square brackets, [ ], are optional. Only one Jam File can be processed at a time. The function of each flag is listed in *Table 1*.

Table 1. Jam Player Options

Flag	Definition	Function
-h	Help	Reports the Jam Player version.
-v	Verbose	Reports status and error messages with detailed real-time information.
-m	Memory	Specifies the amount of memory the Jam Player can use (in bytes).

Checking the CRC

The Jam Player can test the integrity of a Jam File by comparing the actual CRC value of the Jam File to the expected CRC value. The CRC statement, found at the end of the Jam File, stores the expected CRC value. The CRC statement and expected value is calculated and added to the Jam File during the files generation. “Hand-edited” Jam Files may omit the CRC statement. Including the CRC statement and value in the Jam File is optional. The CRC check only occurs when Jam Files include the CRC statement. The CRC check has three possible outcomes: (1) successful match, (2) unsuccessful match, or (3) no CRC statement found. For detailed information on calculating the CRC, refer to the *Jam Programming & Test Language Specification.*

Extracting NOTE Field Information

NOTE fields are stored in a Jam File as pairs of strings: the first of which is the key, and the second of which is the value. The key string must be 32 characters or less in length with the first character being alphabetic and subsequent characters alphanumeric or underscore, and is case-insensitive. The value string can contain any printable ASCII characters (alphabetic, numeric, punctuation, and white-space characters). The only restriction is that double quotation marks (“) must enclose colons (:), semicolons (;), or comment characters (single quotation mark). The value string cannot contain double quotation mark characters. A semicolon terminates the value string. The software interface for extracting NOTE information supports the query of a NOTE value given the key, and extraction of all NOTE fields in the Jam File. The following are examples of NOTE fields:

```
NOTE DEVICE “CY7C37256”;
NOTE CREATOR “CYPRESS SEMICONDUCTOR”;
NOTE JAM_VERSION “1.0”;
NOTE ALG_VERSION “1.1”;
```

Jam Player Output

When the Jam Player encounters an EXIT statement, it terminates executing the Jam File and can output information.

The following are Jam Player outputs:

- Instruction, data, and control information to drive the JTAG hardware interface.
- Optional text messages, which can be displayed on an output device (if available).
- Information exported using the EXPORT statement.
- An EXIT code returned at the termination of processing.

EXPORT Statement

The EXPORT statement transmits a key string and an integer value outside the Jam interpreter to the calling program. The interpretation of the value depends on the key string. *Table 2* lists the export key strings that are defined.

Table 2. EXPORT Key Strings

Key String	Value
PERCENT_DONE	Percent of program executed so far (range 0–100)
UESCODE	Integer value of JTAG UESCODE
IDCODE	JTAG manufactures 32-bit ID

EXIT Codes

EXIT codes are the integer values used as arguments for the EXIT statement. These codes are used to indicate the results of executing a Jam File. An EXIT code value of zero indicates success, while a non-zero value indicates a failure, and identifies the general type of failure that occurred. EXIT codes are not used to indicate errors in the processing of the Jam File, Jam processing errors (such as “Divide by zero” or “Illegal variable name”) are detected and reported by the system which is processing the Jam File. EXIT codes indicate the status of a Jam File which has run to completion, including successful processing of the EXIT statement itself (which terminate the execution of the program). *Table 3* shows the EXIT codes that are defined.

Table 3. EXIT Codes

EXIT Code	Description
0	Success
1	Illegal flags specified in initialization list
2	Unrecognized device ID
3	Device version is not supported
4	Programming failure
5	Blank-check failure
6	Verify failure
7	Test failure

Installing the Jam Player

Software Interface

The Cypress Jam Player communicates with other software using a ‘C’ language function-call interface. Three “entry point” functions can invoke the Jam Player from outside the Jam Player software. In addition, the Jam Player can call eight “callback” functions. These functions provide access to external resources, such as the Jam program and the JTAG hardware interface. To install the Jam Player into a computer system or embedded microprocessor-based system, these callback functions must be customized as appropriate to provide access to the resources of that system.

Hardware Interface

The Jam Player controls the signals of the JTAG hardware exclusively through a callback function called `jam_jtag_io`, which is described in detail below. This function must be customized to permit access to the JTAG hardware interface in the host system. Similarly, the signals of the non-JTAG hardware interface are controlled through a callback function called `jam_vector_io`.

Entry Point Functions

The three entry point functions corresponding to the three operating modes of the Jam Player are as follows:

- `jam_execute`
- `jam_check_crc`
- `jam_get_note`

`jam_execute` *Entry Point Function*

The `jam_execute` function executes a Jam program. The function prototype is as follows:

```
int jam_execute(
  char **init_list,
  char *workspace,
  long size,
  long *error_line,
  int *exit_code);
```

The `init_list` parameter is the address of a table of string pointers, each of which contains an initialization string. The table is terminated by a NULL pointer. Each initialization string is of the form “name=value”. The name should be the name of an integer or Boolean variable in the Jam program, and the value is the numeric value to be assigned to that variable, which overrides the initialization value specified in the Jam program. If no initialization list is needed, then a NULL pointer may be used to signify an empty initialization list.

The `workspace` parameter is a pointer to a memory buffer, to be used for storage of variables and other information by the Jam Player.

The `size` parameter is the number of bytes available in this buffer. If the workspace buffer is too small, an error will occur.

The `error_line` parameter is a pointer to a long integer that receives the line number at which an error occurred, if any. (The occurrence of an error is indicated by the return code from the function.)

The `exit_code` parameter is a pointer to an integer that receives the value of the exit code specified in the EXIT statement, which terminated the Jam program. This exit code is valid only if the Jam file is processed successfully, as indicated by the return code from the function. An exit code of zero indicates that the Jam program has had a successful result (for example, a device was programmed successfully). A list of the EXIT codes is shown in *Table 3*.

The return code from `jam_execute` is an integer that indicates whether the Jam program completed successfully, or was terminated prematurely due to an error. A return value of zero indicates success, and non-zero values indicate error conditions. The error codes are shown in the Jam Player source code file, jamexprt.h.

`jam_check_crc` *Entry Point Function*

The `jam_check_crc` function checks the CRC of a Jam program, without executing the program. The function prototype is as follows:

```
JAM_RETURN_TYPE jam_check_crc(
  unsigned short *expected_crc,
  unsigned short *actual_crc);
```

The `expected_crc` parameter is a pointer to an unsigned short integer that receives the CRC value stored in the CRC statement at the end of the Jam program.

The `actual_crc` parameter is a pointer to an unsigned short integer that receives the CRC value calculated by processing the contents of the Jam program. These parameters permit the CRC values to be displayed in an error message, if desired.

The return code from the `jam_check_crc` function is an integer that indicates the result of the CRC check. A zero indicates a match, while a non-zero value indicates that the CRC values do not match, or that no CRC statement was found in the Jam program.

`jam_get_note` *Entry Point Function*

The `jam_get_note` function extracts NOTE information from a Jam program, without executing the program. This function may be used either to search for a NOTE field with a given key string, or to retrieve all NOTE fields from the Jam program regardless of the key. The function prototype is as follows:

```
JAM_RETURN_TYPE jam_get_note(
  long *offset,
  char *key,
  char *value,
  int length);
```

To obtain the note value for a given key, the key string is provided for the `key` parameter, NULL is passed for the `offset` pointer, and a buffer of sufficient length is provided for the `value` parameter, with the `length` parameter set to the buffer length. This combination causes the entire Jam File to be searched for a NOTE field with a matching key. If such a NOTE is found, the `value` string is copied to the `value` buffer (up to `length` bytes), and a zero return code is returned. A non-zero return code is returned if the NOTE key was not found.

To extract all NOTE fields, a buffer of at least 33 characters must be provided for the `key` parameter (32 key characters + NULL character). The `offset` parameter is a pointer to a long integer that stores the current position in the Jam File, beginning with zero. The function can be called multiple times, each time yielding a NOTE `key/value` pair, until the function returns an unsuccessful (non-zero) return code. Each time the function is called, it saves the position of the end of the previous NOTE statement in the long integer pointed to by `offset`.

Customizing the Jam Player

To customize the Jam Player to a specific platform or application, file I/O and port configurations can be modified by changing the jamstub.c file. When the Jam Player is receiving data, the jamstub.c file can retrieve data from the Jam File, or read data shifted from the TDO pin. The jamstub.c file can also send processed JTAG data to the three JTAG pins (TDI,

TMS, and TCK) and send error and information messages back to the calling program.

Callback Functions

The I/O interfaces of the Jam Player are encapsulated in a set of customizable callback functions. This approach makes the program portable to a wide variety of systems with different I/O interface requirements.

The interface for reading data from the Jam File is encapsulated in two functions that must be customized for the target application. The two interface functions for Jam program input are `jam_getc` and `jam_seek`.

`jam_getc` *Callback Function*

The customizable function `jam_getc` gets a single character from the current position in the Jam program. The return value is the character code of the character that was read, or (-1) if no character is available—for example, if the end of the file was reached. Each call to `jam_getc()` advances the current position in the file, so that successive calls get sequential characters from the file. This behavior is similar to that of the standard 'C' function `fgetc()`. The function prototype is as follows:

```
int jam_getc(void);
```

`jam_seek` *Callback Function*

The customizable function `jam_seek` sets the current position in the Jam File input stream. The function returns zero for success, or a non-zero value if the request offset was out of range. This behavior is similar to that of the standard 'C' function `fseek()`. The function prototype is as follows:

```
int jam_seek(long offset);
```

The storage mechanism for the Jam program can be a file system or a memory buffer. Note that if a file system is used, the equivalent of the 'C' language `fopen()` and `fclose()` functions, as well as storage of the file pointer, are not managed by the Jam Player and must be included in the customization code.

`jam_jtag_io` *Callback Function*

The JTAG hardware interface is implemented using a single customizable function. This function is as follows:

```
int jam_jtag_io(int tms_tdi);
```

This function provides access to the signals of the JTAG hardware interface. Each time this function is called, the logic levels of the JTAG TMS and TDI output signals are set to the requested values, the TDO input signal from the JTAG hardware is sampled and returned in the return code, and then the TCK clock signal is strobed (HIGH, then LOW). The TRST signal is not used. The `tms_tdi` parameter contains three bits of information, indicating the state of the TMS signal and the TDI signal, and also whether the TDO output level should be read. (This parameter allows you to skip the reading of TDO if the value is not going to be used.) The least significant bit (mask 0x01) represents the TMS level, the next bit (mask 0x02) represents the TDI level, and the third bit (mask 0x04) indicates whether the TDO level should be read from the hardware: if set, the TDO value must be read from the hardware, otherwise it is not necessary to read the TDO value. The return code is zero if TDO was LOW, non-zero if TDO was HIGH. If the actual TDO level was not read, zero should be returned.

`jam_message` *Callback Function*

If a console or teletype output device is available, it can be used to display the messages generated by PRINT statements. If no output device is available, PRINT statements will be ignored. The message function interface is implemented using the following function:

```
void jam_message(char *message_text);
```

The Jam Player does not append a new line character at the end of the message string, so for some applications it may be appropriate to do so inside the `jam_message()` function. If the standard 'C' language console output functions are available, the function `puts()` can be used for text message output. If no message device is available, this function should simply do nothing, then return.

`jam_delay` *Callback Function*

The following customizable function can be used to make accurate delays in real time:

```
void jam_delay(long microseconds);
```

This function can use a software delay loop that is calibrated for the speed of the particular target system, or it can make reference to a hardware timer device in the target system to measure the passage of time. For successful device programming, this function must perform accurately over the range of one millisecond (1000 microseconds) to one second (1,000,000 microseconds), with a tolerance of ±1 millisecond. Of course, any error in the positive direction will increase the total programming time for a device programming application.

`jam_export` *Callback Function*

The `jam_export` function is a customizable function that transmits information from the Jam Player to the calling program in the form of a text string and an associated integer value. The text string is called the key string, and it determines the significance and interpretation of the integer value. The syntax is as follows:

```
void jam_export(char *key, long value);
```

The `jam_export` function should ignore any invocation that does not carry a recognized key string.

`jam_vector_map` & `jam_vector_io` *Callback Functions*

Two customizable functions are provided to support non-JTAG hardware access. These functions correspond to the optional VMAP and VECTOR statements in the Jam language. The Cypress FLASH370i devices require a super voltage on the ISREN pin to enable the programming operations, but must be removed after programming to prevent breakdown problems on the Vpp pin. (Refer to "An Introduction to In System Reprogramming with FLASH370i™" application note for a more detailed discussion of FLASH370i ISR.) These functions can be used to control this super voltage. Certain Ultra37000 devices have a JTAGEN pin which must be driven high to place the part into ISR mode. These functions can be used to control the JTAGEN pin. (Refer to "An Introduction to In System Reprogramming with Ultra37000™" application note for a more detailed discussion of Ultra37000 ISR.)

The prototypes for these functions are as follows:

```
int jam_vector_map(
  int signal_count,
  char **signals);
```

```
int jam_vector_io(
  int signal_count,
  long *dir_vect,
  long *data_vect,
  long *capture_vect);
```

The `jam_vector_map` function is used to establish a mapping between signal names and vector bit positions for non-JTAG signals. The order of the signal names in the `signals` array determines the order of bits in the vectors processed by the `jam_vector_io` function. The `jam_vector_map` function should match these signal names to the names of non-JTAG signals available in the host system, and store this mapping.

The `jam_vector_io` function applies a vector to the non-JTAG signals in the current vector mapping (the signals identified in the most recent call to `jam_vector_map`). The number of signals (`signal_count`) must be identical to the signal count specified in the most recent call to `jam_vector_map`. When `jam_vector_io` is called, the direction and data of non-JTAG signals is updated as follows:

If the corresponding bit of `dir_vect` is zero, then that signal will be three-stated (high-impedance).

If the bit is one, then that signal will be made to drive the logic level specified in the corresponding bit of `data_vect`.

If the pointer to `capture_vect` is a NULL pointer, then no information is read back from the non-JTAG signals.

If the pointer to `capture_vect` is not NULL, then for each signal whose direction bit is zero, the current logic level of that signal is sensed and stored in the corresponding bit of `capture_vect`.

All three vector pointers are pointers to arrays of long integers. The bit positions in these vectors are defined as follows: bit zero is the least significant bit of the first long integer addressed by the pointer; bit 31 is the most significant bit of the first long integer; bit 32 is the least significant bit of the second long integer; and so on.

The Jam code for the VMAP and VECTOR statements used in the Jam files for both the FLASH370i and the Ultra37000 devices is shown below.

Beginning of Jam File (enables ISR mode):

```
VMAP "ISRVPP";
VECTOR 1, 0;
```

End of Jam File (disables ISR mode):

```
VECTOR 1, 1;
```

The first argument in the `VECTOR` statement corresponds to the `dir_vect` bit. The second argument in the `VECTOR` statement corresponds to the `data_vect` bit. The `VECTOR` statement in the Jam file that enables ISR mode has a `data_vect` bit of 0. For the FLASH370i devices, a super voltage enables ISR mode. A low signal from the Jam player must enable the super voltage to the ISRen pin of the devices. For the Ultra37000 devices, an active HIGH on the JTAGen pin enables ISR mode. A LOW signal from the player enables ISR mode, and must be inverted to apply a HIGH signal to the devices.

Memory Usage

The Jam Player uses memory in a simple and predictable way, in order to be compatible with embedded systems that lack a memory-allocation service. The Jam Player needs both program storage and dynamic memory. Program storage, such as a hard drive or ROM, is used to store the Jam Player binary and the Jam File. Dynamic memory, such as RAM, is used when the Jam Player is called.

The Jam Player uses memory as follows:

- The embedded processor calls the Jam Player from ROM.
- The Jam Player reads the Jam File from ROM and stores it in RAM.
- The Jam Player inflates the uncompressed programming data contained in the Jam File and stores it in RAM.
- The Jam Player initializes the symbol table, stack, and heap in RAM.

The symbol table holds labels and variables in the Jam File. The stack is used by FOR loops, CALL, and PUSH statements. The heap is temporary memory used for evaluating arithmetic expressions and storing padding data. The Jam Player then parses and executes the Jam File. While processing the Jam File, the stack and heap expands and shrinks in memory as commands are encountered. The amount of dynamic memory needed for the Jam File, the uncompressed data, and the symbol table remain constant during the whole process. The Jam Player memory usage is shown in *Figure 5*.

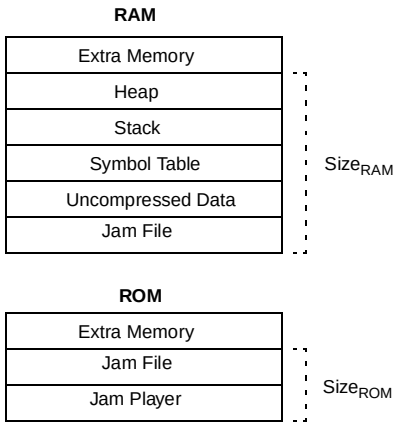


Figure 5. Jam Player Memory Usage

Program Storage (ROM) Usage

The maximum amount of ROM (or program storage) needed can be estimated by:

$Size_{ROM} = Size_{Jam\ Player} + \Sigma\ Size_{Jam\ Files}$

The size of the Jam Player will depend on the embedded processor used and the complexity of the interface logic. The total amount of ROM space required for the Jam Files will depend on the number of devices in the JTAG chain that are being programmed. If there is only one device in the chain, then the required ROM space is the size of Jam File for that device. The size of the Jam File will depend on the target device, and range from 26 Kbytes to 30 Kbytes, using compressed data. The exact size of a Jam File can be determined after running the Cypress Jam Composer. If there are three device in the JTAG chain, for example, and all three of the devices are going to be programmed, the required ROM space is the sum of the three Jam Files.

The approximate sizes of the Jam Player and a single Jam File is shown in *Table 4.*

Table 4. ROM Memory Requirements (single device)

	Typical Size (Kbytes)
Jam Player	90–100
Jam File	26–30
Total ROM	**116–130**

Dynamic Memory (RAM) Usage

The maximum amount of RAM (or dynamic memory) needed can be estimated by:

$Size_{RAM} = \Sigma\ Size_{(Jam\ File\ +\ Uncompressed\ Data)} + Size_{Symbol\ Table}$

The total amount of RAM space required for the Jam Files is the same as required for ROM memory, discussed above in *Program Storage (ROM) Usage*.

After the Jam Player reads the Jam File from ROM and stores it in RAM, the Jam Player inflates the uncompressed programming data contained in the Jam File and stores it in RAM. RAM space required to store the uncompressed data for each Jam File can be determined be looking at the size of the ACA variable in the Jam File. Each ACA variable is listed in the Declaration/Initialization section. The size of each array can be found by looking at the number within the square brackets of the variable declaration statement. For instance:

```
BOOLEAN DATA_IN[434460] = ACA aGD00u@...
```

The size of the ACA variable after inflation is 434460 bits, or approximately 53 Kbytes.

$Size_{Symbol\ Table}$ is the size of the symbol table and is given by the following equation:

$Size_{Symbol\ Table} = 48 \times JAM\_C\_MAX\_SYMBOL\_COUNT$

The size of a variable or label name is 48 bytes. JAM_C_MAX_SYMBOL_COUNT is defined in the jamdefs.h file with a default value of 1021. Practically, most Jam Files will use a maximum of 400 variable and label names. Changing JAM_C_MAX_SYMBOL_COUNT to around 400 can save memory in your embedded system.

$Size_{Symbol\ Table} = 48\ bytes \times 400 = 18.75\ Kbytes$

The stack and the heap require very little memory compared to the total amount of memory used by the Jam Player. The constant JAMC_MAX_NESTING_DEPTH in the jamdefs.h file defines the maximum stack depth.

The approximate RAM requirements for a single Jam File is shown in *Table 5*. The smaller numbers are for the 37032 while the larger numbers are for the 37512.

Table 5. RAM Memory Requirements (single device)

	Typical Size (Kbytes)
Jam File	26–30
Uncompressed Data	2.2–53
Symbol Table	19
Total RAM	**47.2–102**

Conclusion

In system reprogrammability has several benefits. One of the benefits is the ability to reprogram programmable logic devices after being soldered onto a printed circuit board. Systems containing an embedded processor can use the embedded processor to program a chain of ISR devices using the Jam Programming and Test Language.

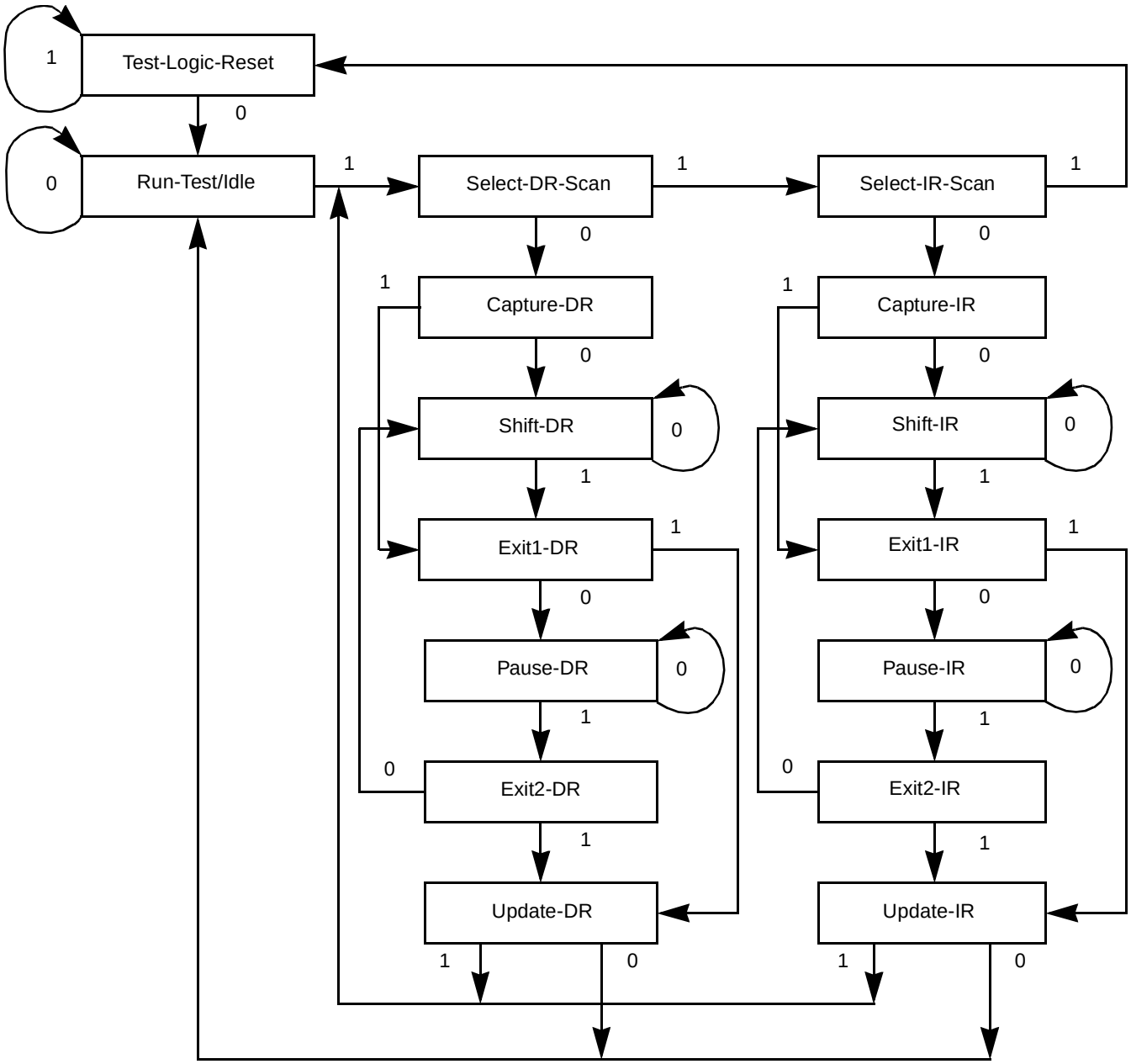
This application note provides information on how to use the Jam language to program ISR programmable devices using an embedded processor. It includes information on the Jam language, the Jam player source code which is customized for different system architectures, hardware considerations, and memory usage.

Appendix A. Low-level Jam Player Operation

JTAG TAP Controller State Machine

Although detailed knowledge of the ISR internal architecture is not required for users to program ISR programmable devices through embedded processors, the low-level implementation of the IEEE 1149.1 (JTAG) Test Access Port (TAP) state machine is provided here. The TAP controller is a 16-state state machine clocked on the rising edge of TCK. The TMS pin is used to control the JTAG operations in the ISR device. See *Figure 6* for a state diagram of the TAP controller.

The Jam Player has a low-level driver that controls the TAP controller. The Jam File contains high-level instructions which cause either the data register branch or the instruction register branch of the TAP state machine to be executed. The high-level instructions are DRSCAN, IRSCAN, and WAIT. DRSCAN scans the JTAG data register. IRSCAN scans the instruction register. WAIT causes the state machine to wait for a certain period of time in state Run-Test/Idle, unless a different state is specified in the WAIT statement.



Note: The “0” and “1” on the transition arrows are the state of signal TMS. Signals are clocked on the rising edge of TCK.

Figure 6. JTAG TAP Controller State Diagram

Appendix B. Functional Behavior of the Jam Player

Functional Behavior of the Jam Player

When the Jam Player encounters instructions DRSCAN, IRSCAN, or WAIT, it controls the output on TCK, TMS, and TDI in order to complete the instruction. *Figure 7* and *Figure 8* show how the Jam Player carries out the DRSCAN, IRSCAN, and WAIT instructions.

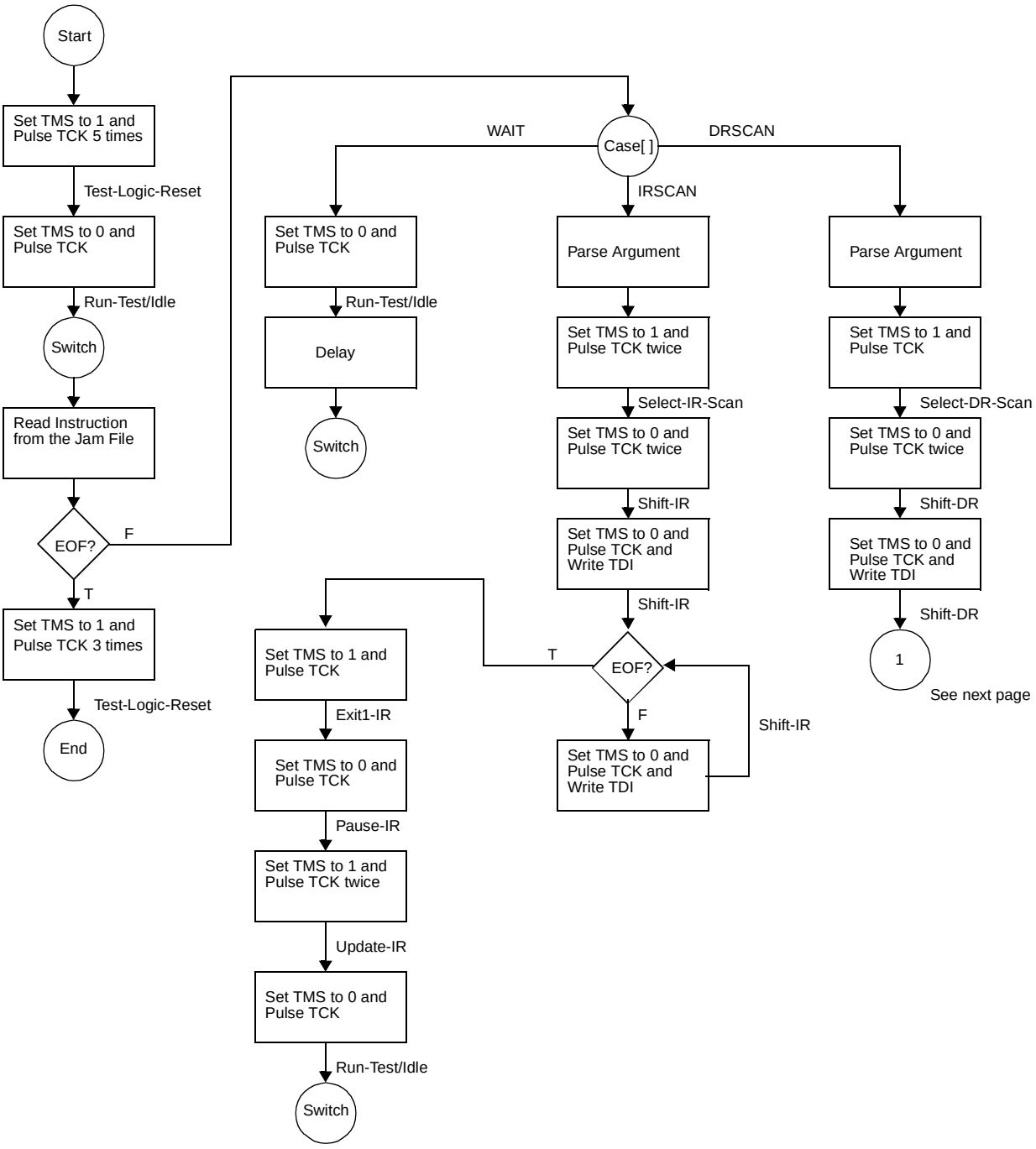


Figure 7. Jam Player—Low-Level Flow Diagram (1 of 2)

Appendix B. Functional Behavior of the Jam Player (continued)

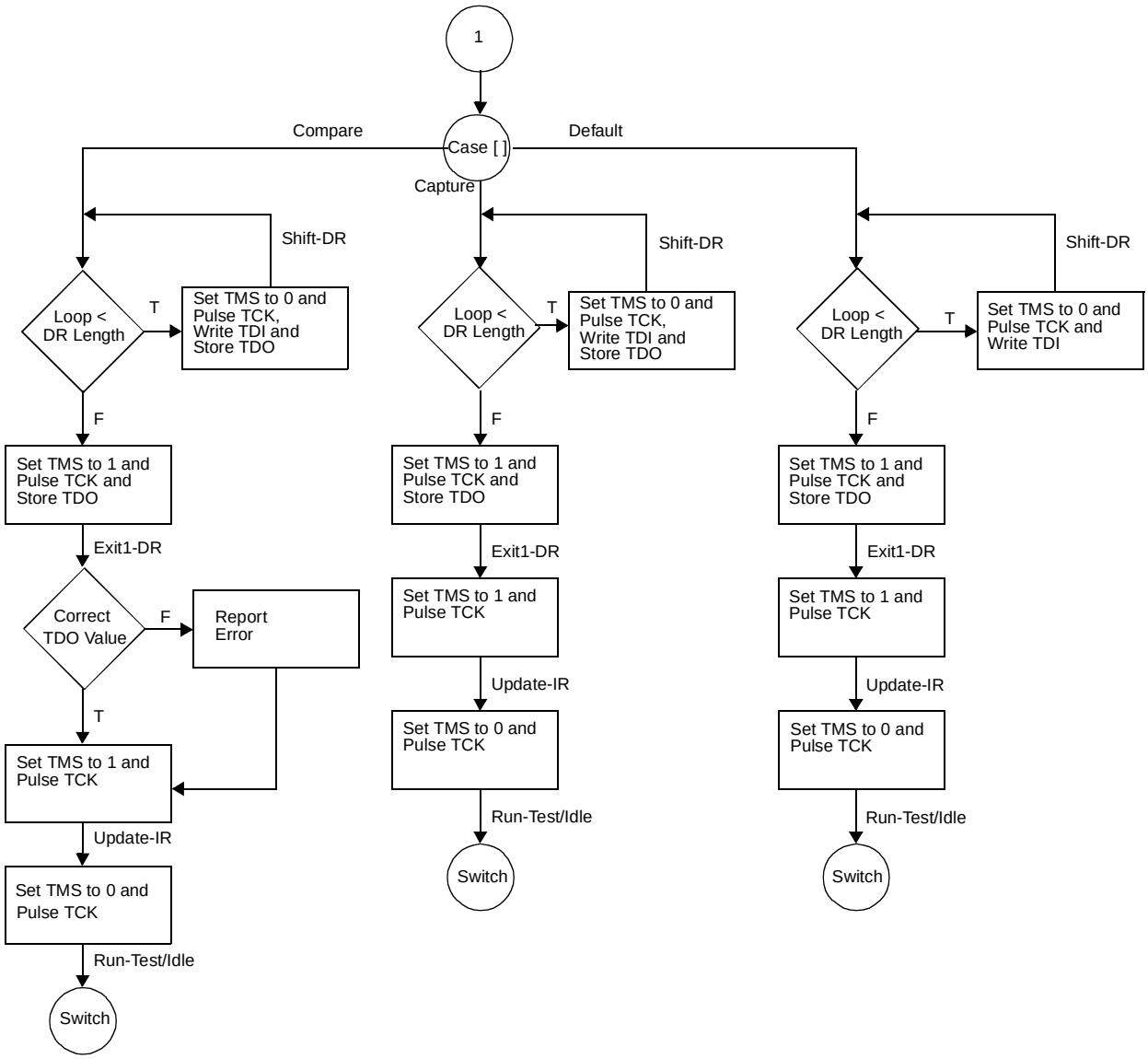


Figure 8. Jam Player—Low-Level Flow Diagram (2 of 2)