Embedded Programming Using the 8051 & Jam Byte-Code

February 1999, ver. 1 Application Note 111

Introduction

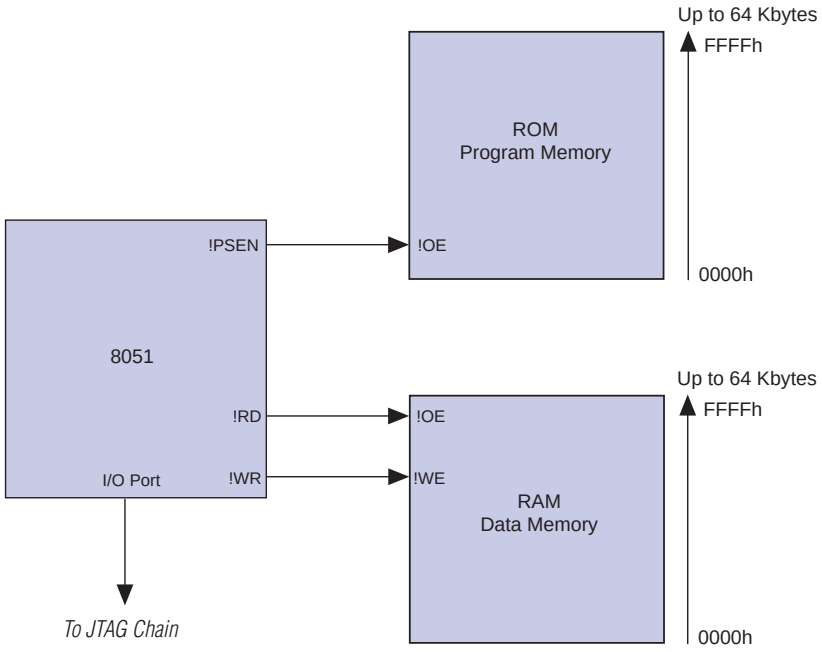
The 8051 family of microprocessors is relatively inexpensive, easy to use, and is a proven platform for managing simple processing tasks. This application note outlines Altera® support for embedded programming and configuration using the 8051 family of microprocessors and a Jam™ Byte-Code File (**.jbc**). Instructions to port the 8051 Jam Byte-Code Player are also provided.

See *Application Note 88 (Using the Jam Language for ISP & ICR via an Embedded Processor)* for more information on supported processors and using Jam Byte-Code.

8051 Architecture

The 8051 architecture consists of separate ROM and RAM addressing. Figure 1 illustrates the 8051 interface as it applies to memory.

Figure 1. 8051 Memory Interface



The 8051 processor can retrieve programming or configuration information from configuration or FLASH devices. The 8051 processor can access up to 64 Kbytes of ROM and 64 Kbytes of RAM, and can be extended by paging additional memory. However, paging memory requires additional discrete logic between the 8051 and associated memory, which slows access and programming times.

The 8051 retrieves and executes instructions from ROM or program memory. Part of executing instructions involves controlling I/O pins that provide access to ROM, RAM, I/O ports, and addresses. For example, when the 8051 retrieves an instruction to access external data, or RAM, the processor automatically toggles the `!RD` pin such that the information in the RAM is retrieved and stored in the appropriate internal registers. These actions are performed automatically by the processor.

Many variants of the basic 8051 architecture exist, including different clock speeds (12 to 50 MHz), 8- or 16-bit functions, and 0 to 24 Kbytes of on-chip ROM. When programming devices that contain more than 64 macrocells, Altera recommends using the fastest 8051 for the best programming times.

Jam Byte-Code Software

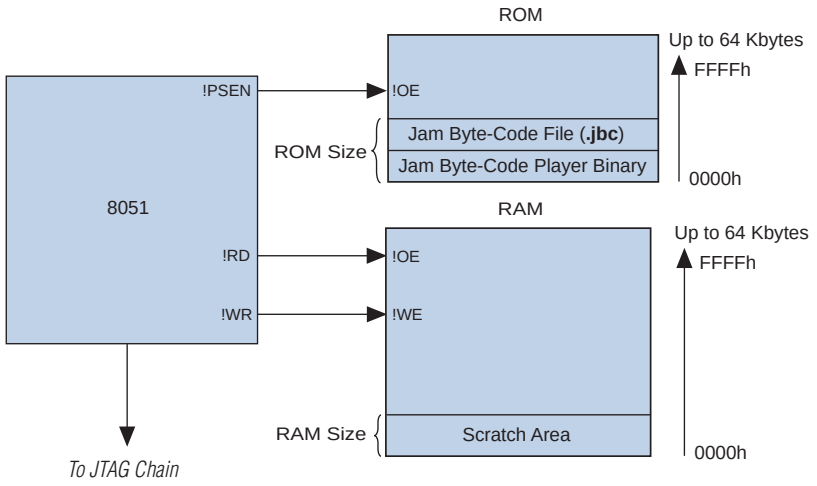
The Jam Byte-Code Player and a JBC File are needed to program or configure Altera devices using the 8051 processor. This source code can be customized for the target 8051. For more information on how to customize source code, see "Porting the Jam Byte-Code Player".

JBC Files can be generated using the MAX+PLUS® II software, or by compiling an existing ASCII-based Jam File (**.jam**) into a Jam Byte-Code equivalent using the stand-alone Jam Byte-Code Compiler.

You can download the compiler at **http://www.jamisp.com**. The Jam Byte-Code Player source code can be obtained by contacting Altera Applications at 1 (800) 800-EPLD, or sending an e-mail to **sos@altera.com**.

Figure 2 shows one way to store the Jam software in an 8051 embedded system.

Figure 2. 8051 Architecture



Although Figure 2 shows the JBC File stored in ROM, the file could also be stored and executed in RAM. In either case, the Jam Byte-Code Player must be executed by the 8051 processor, and must have access to the JBC File.

Porting the Jam Byte-Code Player

The 8051 Jam Byte-Code Player is written in the C programming language for the 8051 architecture. The source code is provided in the **jbi51.c** file, which is provided with the player and is designed to make porting as easy as possible. Table 1 shows the source code's default configuration.

Table 1. 8051 Jam Byte-Code Player Configuration Options

Configuration	Processor	Compiler	JBC File Storage
Default	Dallas DS87C520	Keil	ROM
Other	Any 8051 Variant	Any 8051 C Compiler	ROM, RAM, Or Serial Port *(1)*

Note:
(1) Programming can be accomplished by downloading the JBC File from remote storage to local RAM via the serial port.

The source code can be compiled for any 8051 variant as long as the supporting compiler can compile C code. If any of the target system's parameters differ from the default, the Jam Player source code can be customized to accommodate them, including storage options and execution of the Jam Byte-Code Player binary and JBC File. Porting the source code requires three basic steps.

Step 1: Edit Compiler-Specific Keywords

If a compiler other than the Keil compiler is used, the keywords in Figure 3 must be changed at the beginning of the **jbi51.c** file:

Figure 3. Compiler-Specific Keywords

```
#define CONSTANT_AREA code
#define XDATA_AREA xdata /* external RAM accessed by 16-bit pointer */
#define PDATA_AREA pdata /* external RAM accessed by 8-bit pointer */
#define IDATA_AREA idata /* internal RAM accessed by 8-bit pointer */
#define RDATA_AREA data  /* internal RAM registers with direct access */
#define BIT bit          /* internal RAM single bit data type */
```

The Keil keywords are `code`, `xdata`, `pdata`, `idata`, `data`, and `bit`. The `code` keyword refers to program memory where the Jam Byte-Code Player binary is stored. All other keywords refer to specific internal and external RAM spaces. These memory spaces are specific to the 8051 architecture. Change these keywords to map to the compiler you are using.

Step 2: Customize the Memory Map

You can store the JBC File in either ROM or RAM. However, you must customize the Jam Player according to the JBC File location. The JBC File's default location is in ROM.

ROM

To specify ROM as the JBC File's location, you must set the global variable, using the following method.

1. Convert the binary JBC File into hexadecimal data, and set `jbi_program[]` equal to the hex data at the end of the **jbi51.c** file.

```
#ifndef JBC_FILE_IN_RAM
unsigned char CONSTANT_AREA jbi_program[] =
{
    0       /* insert JBC program data here */
};
#endif /* JBC_FILE_IN_RAM */
```

A program called **jbc2data**, which is provided on the Jam web site at **http://www.jamisp.com**, is used to convert the binary JBC File into a hexadecimal array. The compiler automatically links the JBC information with the binary. If needed, you can use the linker to specify the exact storage location within the ROM.

RAM

Loading the JBC File into RAM requires two steps:

1. Add the following line to the beginning of the **jbi51.c** file.

```
#define JBC_FILE_IN_RAM
```

2. Load the JBC File into RAM before `jbi_execute()` is called by copying the JBC File from its source to the `jbi_program[]` array. The `jbi_program[]` array points to the JBC File as it is stored in RAM. The Jam Player accesses the JBC File through `jbi_program[]`. By default, the code in `main()` reads the JBC File from the 8051 serial port into the `jbi_program[]` global array. The provided code was used to communicate between a PC and the 8051 via the serial port. You may need to customize the code depending on your actual target system.

Step 3: Customize the I/O Routines

The 8051 Jam Byte-Code Player source code was written so that all I/O functions are confined to a few routines. These routines may require customization based upon the system-level hardware. Table 2 shows each routine and the corresponding I/O function.

Table 2. Routines & Corresponding I/O Functions

Routine	Function
`jbi_jtag_io()`	Interface to the IEEE Std. 1149.1 JTAG signals `TDI`, `TMS`, `TCK`, and `TDO`.
`jbi_message()`	Prints information and error text to standard output, when available.
`jbi_export()`	Passes information such as the user electronic signature (UES) back to the calling program.
`jbi_delay()`	Implements the programming pulses or delay needed during execution.

jbi_jtag_io()

This routine is the interface to the IEEE Std. 1149.1 (JTAG) signals `TDI`, `TMS`, `TCK`, and `TDO`. By default, the IEEE Std. 1149.1 signals are mapped to the hardware ports in Table 3.

Table 3. JTAG Mapping Hardware Ports

Signal	Hardware Port
`TDI`	`P1.0`
`TMS`	`P1.1`
`TCK`	`P1.7`
`TDO`	`P3.5`

These signals can be remapped, depending on the hardware port and pins used. The actual pin names are accessed with keywords defined in the library that supports the targeted 8051 processor. By default, the source code calls for the Dallas DS87C520. The pins of each port in the DS87C520 are designated by *P<port number>_<pin number>*. For example, `TDI` is mapped to port 1, pin 0, which is designated `P1_0`. The ports and pins should be remapped based upon the convention of the library used with the targeted 8051.

You must preserve the write and read sequence to and from the ports within this routine. Disruption of the write and read process results in Jam Player errors.

jbi_message()

The `jbi_message()` routine prints information and error messages to standard output. In most applications you will not use this function, so you can either remove the routine or comment out the call to `puts()`.

jbi_export()

The `jbi_export()` routine returns information from the Jam Player to a calling program. The most common use of this routine is to transfer the UES instruction code back to the program that calls the Jam Player. By default, the Jam Player prints the value using `printf`. If `printf` is not available, the UES instruction can be passed back to the calling program, and the calling program must decide whether or not to program the device based on the actual contents of the UES value.

jbi_delay()

Pulses of varying widths are used to program the internal EEPROM cells of Altera MAX devices. The Jam Player uses the `jbi_delay()` routine to implement these pulse widths. This routine must be customized based on the speed of the processor and the time it takes the processor to execute a single loop. By default, the routine is coded so that the absolute delay time (in microseconds) is divided by eight, which is used as the number of times that the processor loops to achieve the specified delay. The default setting is for an 8051 running at 33 MHz. If the target 8051 does not loop eight times per microsecond, the `count` variable must be adjusted. The `jbi_delay()` routine must perform accurately between the range of one millisecond to one second. The function should not delay more than 10% over the time specified, and it cannot return in less time.

With all three steps completed, the 8051 Jam Byte-Code Player is ready to be compiled and run on the target processor.

Executing the Jam Player

`jbi_execute()` is the main entry point for the 8051 Jam Byte-Code Player. To successfully call and run the Jam Player, `jbi_execute()` must receive the correct information. This routine is called from `main()` by default. The remaining code within `main()` sets up the variables that are passed to `jbi_execute()` and handles errors that may be returned by `jbi_execute()`. The call to `jbi_execute()` is shown below:

```
exec_result=jbi_execute(init_list, &error_address,
  &exit_code);
```

An initialization list tells the Jam Player which functions to perform (e.g., program and verify) and is passed to `jbi_execute()`. Once the Jam Player has completed a task, it returns with an exit code. If there are any errors, `jbi_execute()` returns the location of those errors within the JBC File as an address.

The initialization list must be passed in the correct manner. If an invalid initialization list or no initialization list is passed, the Jam Player simply checks the JBC File. If the syntax check passes, the Jam Player issues a successful exit code without performing the program function. The `init_list` variable is an array of pointers to an array of characters. In other words, `init_list` is a two-dimensional array that is assigned a series of string commands to provide instructions to the Jam Player. For example, you can use the following code to set up the `init_list` to instruct the Jam Player to perform a program and verify operation:

```
char CONSTANT_AREA init_list[][]="DO_PROGRAM=1",
  "DO_VERIFY=1";
```

This code declares the `init_list` variable while setting it equal to the appropriate parameters. `CONSTANT_AREA` is the identifier that instructs the compiler to store `init_list` in program memory. The default code sets `init_list` differently, and is used with a terminal program to give instructions to the Jam Player via a command prompt. In most cases, you will not run the Jam Player with this type of interaction.

Two code types can be returned by `jbi_execute()`. The first code type is returned in the variable `exit_code` and the second code type is returned in the variable `error_code`. These codes indicate the functional result of the operation. These codes flag problems with memory limitations or syntax errors, which indicate software issues specific to the Jam Byte-Code Player or the Jam Byte-Code File. Tables 4 and 5 list the possible codes returned in the `exit_code` and `error_code` variables.

Table 4. exit _code Variable Return Codes

exit _code	Description
0	Success
1	Illegal flags are specified in the initialization list
2	Unrecognized device ID
3	Device version is not supported
4	Programming failure
5	Blank-check failure
6	Verify failure
7	SRAM configuration failure

Table 5. error_code Variable Error Codes

Variable	error _code	Description	Action
JBIC_OUT_OF_MEMORY	1	Call to malloc() in jbi_malloc() failed to allocate dynamic memory.	Check available physical RAM before running Jam and compare with the estimate explained in the "Memory Resources" section.
JBIC_STACK_OVERFLOW	2	Stack requires greater than 128 items.	Increase JBI_STACK_SIZE.
JBIC_TO_ERROR	3	JBC File is corrupt.	Check JBC File against a good file. Replace the file if necessary.
JBIC_UNEXPECTED_END	4	Unexpected end of JBC File.	Check for corrupt file. If intact, contact the vendor who created the JBC File.
JBIC_ILLEGAL_OPCODE	5	Unexpected Byte-Code parameter.	Contact vendor for new file.
JBIC_INTEGER_OVERFLOW	6	Integer value exceeded legal range (32 bits).	Contact vendor for new file.
JBIC_DIVIDE_BY_ZERO	7	Internal error from JBC File.	Contact vendor for new file.
JBIC_CRC_ERROR	8	Contents of JBC File are corrupt.	Regenerate JBC File and replace.
JBIC_INTERNAL_ERROR	9	Unexpected Jam Byte-Code Player execution.	Contact vendor with source code error.
JBIC_BOUNDS_ERROR	10	Error in JBC File–JBC algorithm specifies incorrect bounds on array size or other parameters.	Contact vendor for new JBC File.
JBIC_VECTOR_MAP_FAILED	11	Jam Byte-Code Player does not support VECTOR command.	Contact vendor for Jam Player that supports the VECTOR command.
JBIC_USER_ABORT	12	Unused.	None.

Each error, except JBIC_OUT_OF_MEMORY, reflects a corrupt or incorrect JBC File, or improper installation of the Jam Player.

Memory Resources

The 8051 Jam Byte-Code Player takes 29 Kbytes of program memory. You can store the JBC File in program memory or load it into data memory just prior to execution of the Jam Byte-Code Player. The JBC File size is dependent on which and how many devices are targeted for programming or configuring, as well as whether the JBC file uses compression. If the JBC File uses compression, it will take some extra time to decompress the file, resulting in a longer programming or configuration time. Altera offers the option to generate JBC Files that do not use compression. To override the MAX+PLUS II default and disable compression, add the following information to the `[system]` section of your **maxplus2.ini** file:

```
SVF_JBC_USE_COMPRESSION=OFF
```

This setting results in shorter programming times but uses more memory because the JBC File will be significantly larger. Each design must be evaluated based upon the available memory resources to determine whether compressed or uncompressed JBC Files should be used. Some high-density devices which require a large programming file cannot be supported due to the 64 Kbyte program memory limitation in the 8051.

See *Application Note 88 (Using the Jam Language for ISP & ICR via an Embedded Processor)* for more information on compressed and uncompressed JBC file sizes.

Programming Times

As PLDs increase in density, performance, and complexity, so do the software tools and algorithms that support them. As a result, the file sizes and algorithms for EEPROM, FLASH, and SRAM-based architectures have increased in complexity. At the same time, the complexity and performance of the 8051 has not changed. Programming and configuration times are sub-standard when compared to other families of processors. You should consider this factor when designing a system that relies on timely in-field upgrades of programmable logic. Densities and complexity of the target logic makes 8051 programming times unsuitable for certain applications. Performance can vary widely as a function of the density of the target logic device and the speed and efficiency of the 8051. Programming times can vary from two minutes, for a 64-macrocell device, to more than 10 minutes for a 256-macrocell device. Altera recommends using the 8051 for in-field upgrades only when reasons for employing an 8051 outweigh the costs associated with long programming times. In these cases, the 8051 variant with the highest performance should be chosen. Altera also recommends using an 8051 for programming product-term device densities below 256 macrocells.

Conclusion

Easy in-field upgrades can be made using the 8051 family of microprocessors and the Jam Byte-Code Player. Porting the Jam Byte-Code player can be accomplished in three steps. Source code is provided to make porting simple and to support the ability to upgrade to a variety of device densities. This source code is specific to the 8051 family of microprocessors and is compatible with any 8051 device variant. You can generate program files easily using the MAX+PLUS II software, which provides control over programming times and memory utilization. The 8051 microprocessor is the ideal tool for programming low-density devices.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com