PROGRAMMING FLASH THROUGH THE JTAG INTERFACE

Relevant Devices

This application note applies to the following devices:

C8051F000, C8051F001, C8051F002, C8051F005, C8051F006, C8051F010, C8051F011, C8051F012, C8051F015, C8051F016, C8051F017, C8051F206, C8051F220, C8051F221, C8051F226, C8051F230, C8051F231, C8051F236, C8051F020, C8051F021, C8051F022, and C8051F023.

Introduction

This document describes how to program the FLASH memory on C8051 devices through the JTAG port. Example software is included at the end of this note.

NOTE: All Silicon Labs devices can be programmed through the JTAG interface. However, the C8051F2xx family of devices does not support the IEEE 1149.1 boundary scan function.

The information required to perform FLASH programming through the JTAG interface can be divided into three categories:

1. JTAG interface information:
   a. The 4-pin physical layer interface (TCK, TMS, TDI, and TDO)
   b. The Test Access Port (TAP) state machine
   c. TAP Reset, Instruction Register Scan, and Data Register Scan primitives
2. JTAG Indirect Register operations:
   a. Reading an indirect register
   b. Writing to an indirect register
   c. Polling the Busy bit to see when the read or write operation has completed
3. FLASH Programming operations:
   a. Read a FLASH byte
   b. Write a FLASH byte
   c. Erase a FLASH page
   d. Erase the entire FLASH

Figure 1 shows the programming hierarchy for accessing the FLASH through the JTAG port.

JTAG Interface

This note provides enough information about the JTAG interface to enable FLASH programming. For more information, the JTAG standard, IEEE

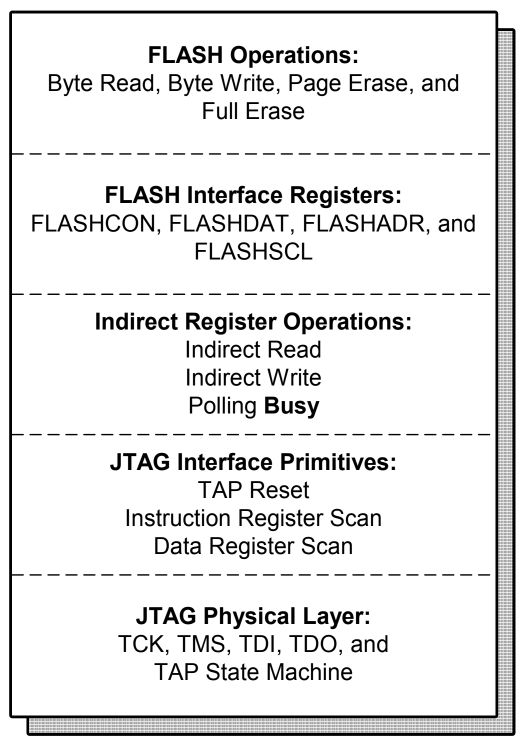

Figure 1. JTAG FLASH Programming Hierarchy

1149.1-1990, can be obtained from the Institute of Electrical and Electronics Engineers (for information, see http://standards.ieee.org). The JTAG interface on C8051 devices is fully compliant with the IEEE 1149.1 specification. Those already familiar with JTAG can skip to the section titled "Instruction Register on C8051 Devices," on page 7.

Test Access Port (TAP) Interface

The hardware interface to the JTAG port consists of four signals, as shown in Figure 2:

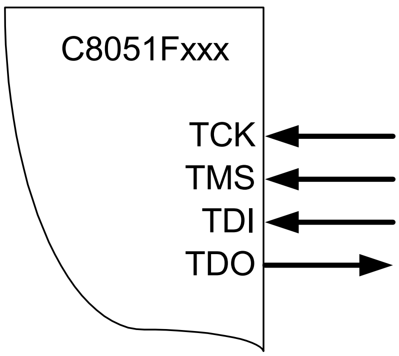

Figure 2. TAP Interface

1. **TCK** input shift clock. Data is sampled at TMS and TDI on the rising edge of TCK. Data is output on TDO on the falling edge of TCK.

2. **TMS** input mode select. TMS is used to navigate through the TAP state machine.

3. **TDI** input. Input data to the Instruction Register (IR) or the Data Register (DR) is presented to the TDI input, and sampled on the rising edge of TCK.

4. **TDO** output. Output data from the Instruction Register or the Data Register is shifted out TDO on the falling edge of TCK.

TAP State Machine

The primary purpose of the Test Access Port state machine, which is shown in Figure 3, is to select which of two shift registers, the Instruction Register or the Data Register, to connect between TDI and TDO. In general, the Instruction Register is used to select which Data Register to scan. The numbers next to the arrows in the diagram refer to the logic state of TMS at the time TCK is brought high.

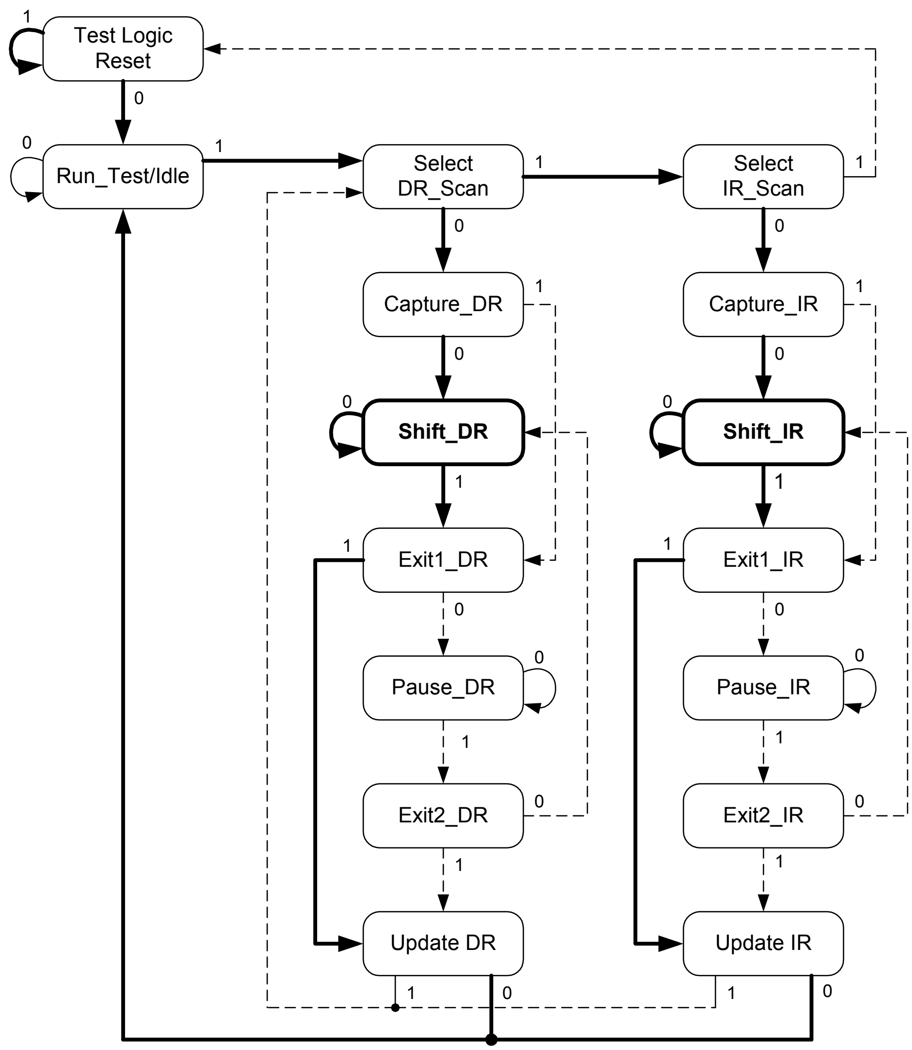

Figure 3. TAP State Machine

TAP Reset

The TAP logic is reset by holding TMS high (logic '1') and strobing (bringing high and then back low) TCK at least five times, as shown in Figure 4.

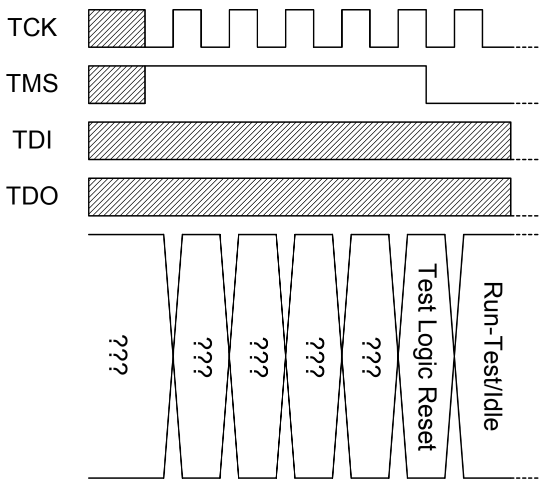

Figure 4. TAP Reset Timing

This advances the state machine to the Test Logic Reset state from any state in the TAP state machine, which resets the JTAG port and test logic. It does not reset the CPU or peripherals.

TAP Notes:

1. Data is valid on TDO beginning with the falling edge of TCK on entry into the Shift_DR or Shift_IR states. TDO goes “push-pull” on this TCK falling edge and remains “push-pull” until the TCK rising edge.
2. Data is not shifted on entry into Shift_DR or Shift_IR.
3. Data is shifted on exit of Shift_IR and Shift_DR.

IR and DR Scan

In addition to test logic reset, there are two primitive operations that the state machine controls: Instruction Register (IR) Scan, and Data Register (DR) Scan. In a scan operation, data is sampled at TDI on the rising edge of TCK, and is output on TDO on the falling edge of TCK. During an Instruction Register Scan operation, the Instruction Register is transferred in the Shift_IR state. During a Data Register Scan operation, the Data Register is transferred in the Shift_DR state. Data is always shifted LSB-first.

In C8051 devices, the Instruction Register is always 16 bits in length. The length of the Data Register varies, depending on the register selected.

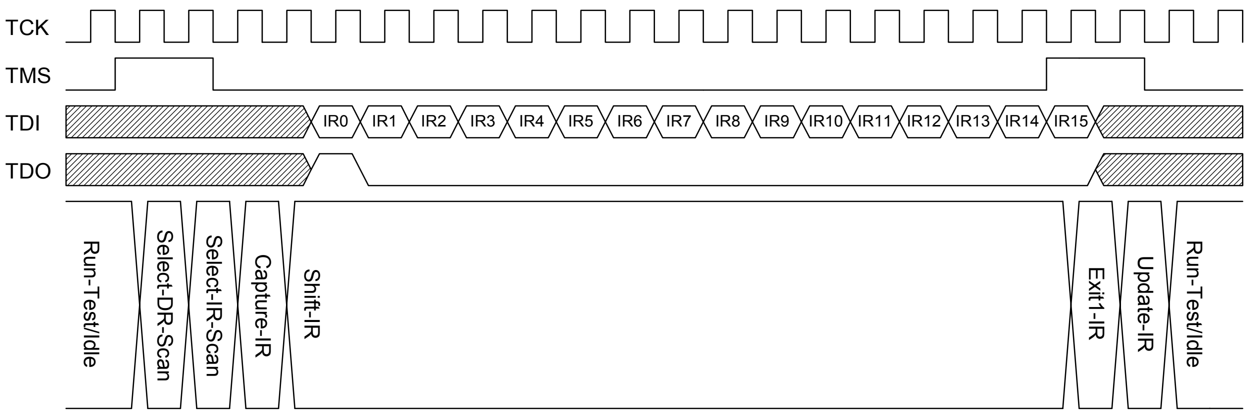

Figure 5. Instruction Register Scan Timing

Figure 5 shows a timing diagram for an Instruction Register access.

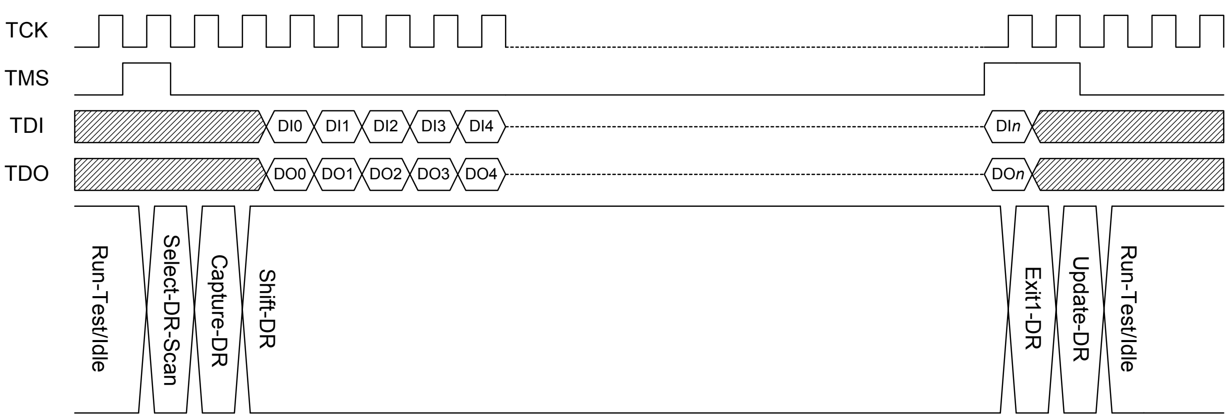

Figure 6. Data Register Scan Timing

Figure 6 shows timing for a Data Register access.

IDCODE Example

To better illustrate how a typical JTAG operation works, we present an example access, in this case, reading the IDCODE register.

Reading the IDCODE is a two-step process. First, an Instruction Register Scan operation is initiated, and the Instruction Register is loaded with the IDCODE address, 16-bits shifted on TDI, as shown in Figure 7. Once the Instruction Register has been loaded, a Data Register Scan operation is initiated, and the 32-bit IDCODE is read from the device, on TDO, as shown in Figure 8.

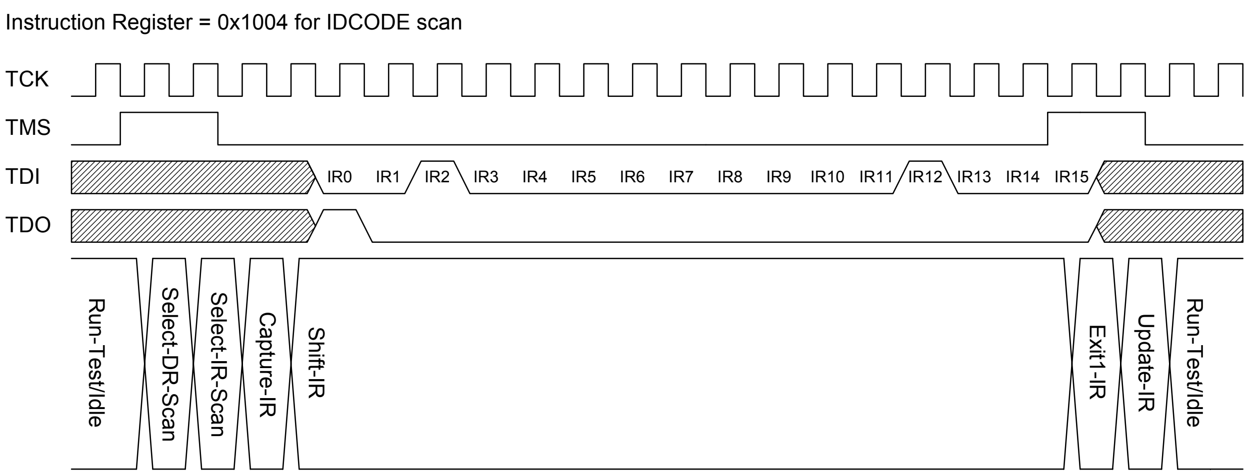

Figure 7. Instruction Register Scan Timing for IDCODE Read

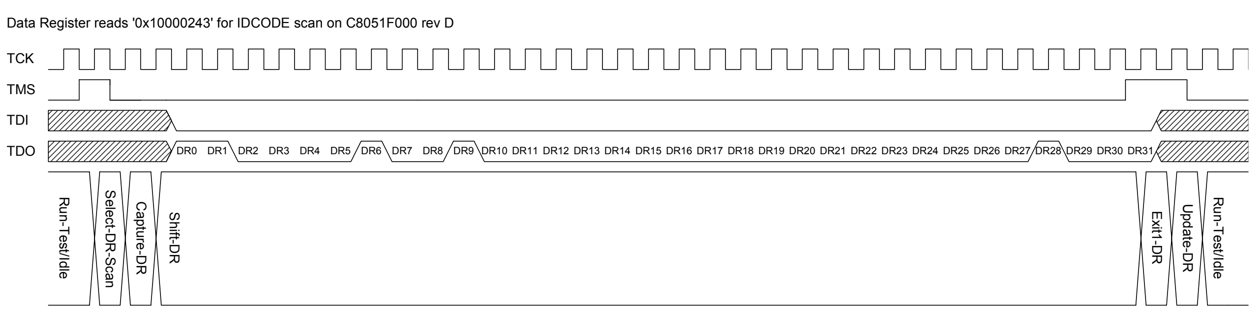

Figure 8. Data Register Scan Timing for IDCODE Read

Instruction Register on C8051 Devices

The Instruction Register (IR) on C8051 devices is always 16-bits in length, and is decoded as follows:

Table 1. Instruction Register Decoding

15:12	11:0
StateCntl	**DRAddress**

The StateCntl field controls the state of the debug hardware. In a FLASH programming operation, the system is first Halted, and then the CPU core is held in Suspend mode to bypass the Watchdog timer.

Table 2. StateCntl Decoding

StateCntl*	Device State
0000	Normal
0001	Halt
0010	System Reset
0100	CPU Core Suspend
1111	Normal
*unlisted states are reserved	

Table 3. DRAddress Decoding

Register	DRAddress*
EXTEST	0x000
SAMPLE/PRELOAD	0x002
IDCODE	0x004
BYPASS	0xFFF
FLASHCON	0x082
FLASHDAT	0x083
FLASHADR	0x084
FLASHSCL	0x085
*unlisted states are reserved	

Indirect Registers

The four FLASH registers (FLASHCON, FLASHADR, FLASHDAT, and FLASHSCL) are accessed using a common indirect method. This indirect scheme handles the information transfer between the JTAG clock domain, controlled by TCK, and the CPU clock domain, controlled by SYSCLK. These FLASH indirect registers are not to be confused with the standard 8051 indirect registers R0 and R1.

Overview of Indirect Register Accesses

To read or write to an indirect register, the Instruction Register is first loaded with the proper **DRAddress**. Reads and writes are then initiated by writing the appropriate Indirect Operation Code (**IndOpCode**) to the selected data register. On a write, the Write opcode is followed by the data to be written.

The format for the data register for the incoming commands is as follows:

Table 4. Indirect Write DR Format

19:18	17:0
IndOpCode	**WriteData**

The Indirect Operation Code (IndOpCode) bits are decoded as follows:

Table 5. IndOpCode Decoding

IndOpCode	Operation
0x	Poll
10	Read

Table 5. IndOpCode Decoding

IndOpCode	Operation
11	Write

The format for the data register for outgoing data is as follows:

Table 6. Indirect Read DR Format

19	18:1	0
0	**ReadData**	**Busy**

Indirect Read

The **Read** operation initiates a read from the register selected by DRAddress. Reads can be initiated by shifting only two bits into the indirect register (the **Read** IndOpCode bits). After the Read operation is initiated, the Busy bit is polled to determine when the operation has completed and the data is available for reading. Figure 9 shows a flow chart that describes how to perform a read operation on an indirect register.

Indirect Write

The **Write** operation initiates a write of **WriteData** to the register selected by DRAddress. Registers of any width up to 18 bits can be written. If the register to be written contains fewer than 18 bits, **WriteData** should be left-justified (MSB occupies bit 17). This allows shorter registers to be written in fewer JTAG clock cycles. For example, a write to an 8-bit indirect register can be accomplished by shifting only 10 bits (2-bit **Write** opcode + 8 data

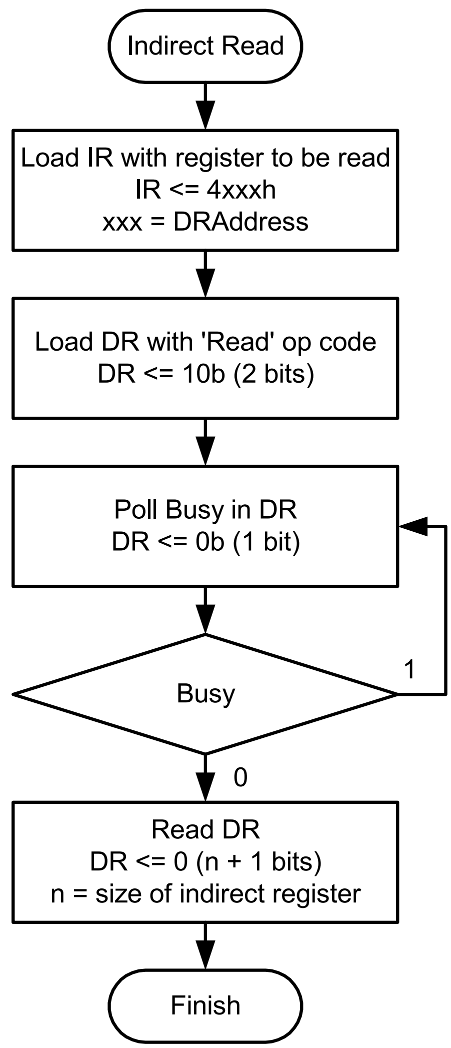

Figure 9. Indirect Read Flow Chart

bits). After a write is initiated, the **Busy** bit should be polled to determine when the operation has completed. Figure 10 shows a flow chart describing how to perform a write operation on an indirect register.

Polling Busy

The **Busy** bit indicates that the current read or write operation has not completed. It goes high ('1') when an operation is initiated and returns low ('0') on completion. Because the Busy bit occupies the LSB of the returned data, polling for **Busy** can be accomplished in one DR shift cycle (on exit of the Shift_DR state).

On an Indirect Read, once **Busy** has gone low, the **ReadData** can be shifted out. Note that the ReadData is always right-justified. This allows registers less than 18-bits to be read in fewer JTAG clock cycles. For example, an 8-bit Read can be performed in 9 DR shifts (8 data bits + 1 Busy bit).

Figure 11 shows the Data Register Scan timing for polling the Busy bit.

The contents of the Instruction Register should not be altered when a Read or a Write operation is in progress.

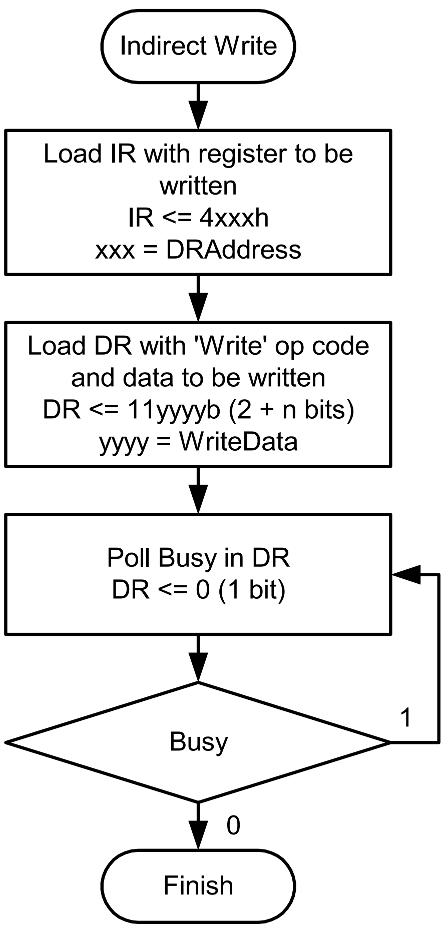

Figure 10. Indirect Write Flow Chart

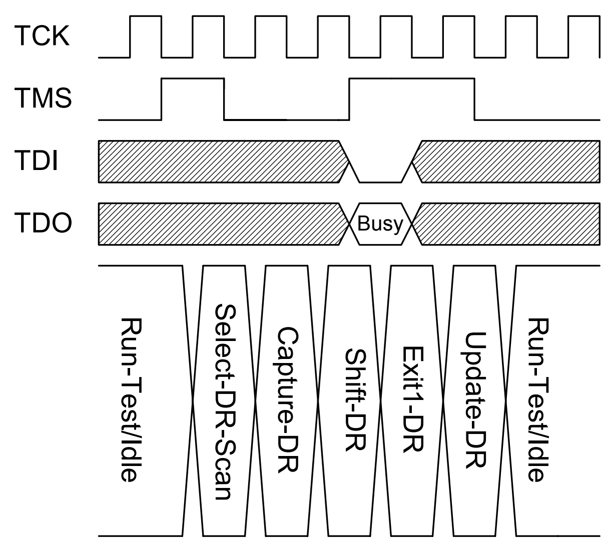

Figure 11. DR Scan Timing for Polling the Busy Bit

FLASH Programming

FLASH Register Descriptions

The FLASH is accessed through four indirect registers: FLASHCON, FLASHADR, FLASHDAT, and FLASHSCL. Each of these registers is accessed using Indirect Read and Indirect Write operations as outlined in the previous section.

FLASHCON

FLASHCON is an 8-bit register that controls how the FLASH logic responds to reads and writes to the FLASHDAT register. The FLASHCON register contains a **ReadMode** setting and a **WriteMode** setting, decoded as follows:

Table 7. FLASHCON Decoding

7:4	3:0
WriteMode	**ReadMode**

Table 8. ReadMode Decoding

ReadMode*	Operation
0000	**FLBusy** Polling
0010	Initiate FLASH read; Increment FLASHADR
*unlisted states are reserved	

Table 9. WriteMode Decoding

WriteMode*	Operation
0000	**FLBusy** Polling
0001	Initiate FLASH Write; Increment FLASHADR
0010	Initiate page erase on current page if FLASHDAT = 0xA5; Initiate erase of entire FLASH if FLASHDAT = 0xA5 and FLASHADR is set to the address of the FLASH Read Lock Byte or the FLASH Write/ Erase Lock Byte.
*unlisted states are reserved	

FLASHADR

FLASHADR is a 16-bit register that contains the address of the FLASH byte to be read or written. FLASHADR is automatically incremented on completion of a read or a write operation.

FLASHDAT

FLASHDAT is a 10-bit register containing 8-bits of data, an **FLFail** bit, and an **FLBusy** bit, as shown below:

Table 10. FLASHDAT Read Decoding

9:2	1	0
FLData	**FLFail**	**FLBusy**

A write to FLASHDAT need only consist of 8 bits because the last bit latched assumes the MSB position.

A read of FLASHDAT requires 11 *DR_SHIFT* cycles (8 for **FLData**, 1 for **FLFail**, 1 for **FLBusy**, and 1 for **Busy**).

Polling for **FLBusy** requires at least 2 *DR_SHIFT* cycles, 1 for **FLBusy** and 1 for **Busy**.

FLASHSCL

FLASHSCL is an 8-bit register that sets the prescale value required for deriving the timing for FLASH operations. When operating from the internal 2 MHz system clock, this register should be configured with 0x86 as follows:

Table 11. FLASHSCL Configuration

7:4	3:0
1000	**0110**

FLASH Access Procedures

Before the FLASH can be programmed, the device needs to be reset and the Watchdog timer taken offline. Otherwise, the Watchdog timer may initiate a system reset during a FLASH operation, resulting in undefined behavior.

Disabling the Watchdog Timer (WDT)

A flow chart showing the process for disabling the Watchdog timer is shown in Figure 12. The procedure is as follows:

1. The system is reset by loading the Instruction Register (IR) with 0x2FFF.
2. An IDCODE scan is performed by loading IR with 0x1004, followed by a 32-bit DR scan with 0x00000000.
3. All following IR addresses set StateCntl to '0x4', which keeps the core in SUSPEND mode, and takes the FLASH off-line.

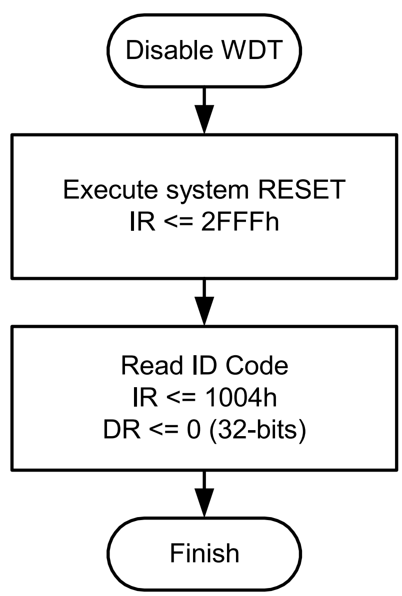

Figure 12. Flow Chart for Bypassing the Watchdog Timer

Reading a FLASH Byte

Figure 13 shows a flow chart which illustrates how to read a FLASH byte. The procedure is as follows:

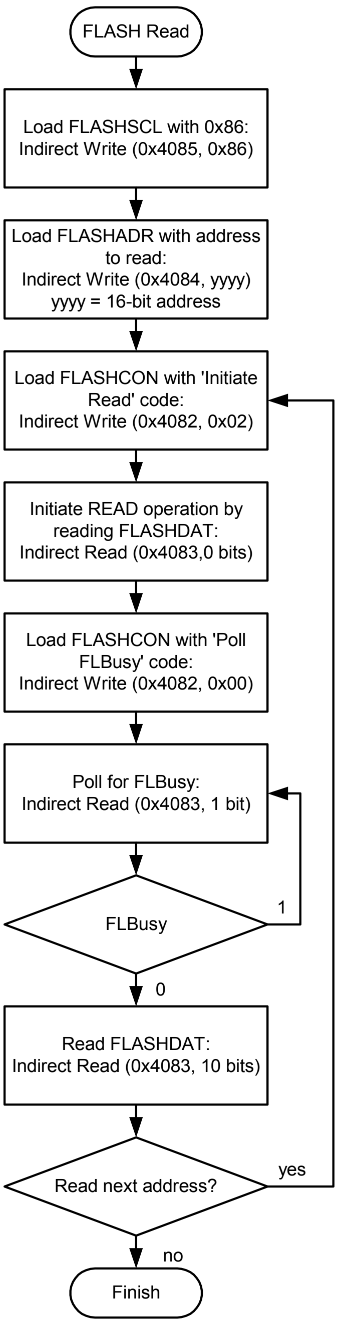

Figure 13. Flow Chart for Reading a FLASH Byte

1. Load FLASHSCL with 0x86, to set proper FLASH timing using the internal 2 MHz system clock. This is accomplished by an Indirect Write of 0x86 to FLASHSCL.

2. Load FLASHADR with the 16-bit address to be read. This is accomplished with an Indirect Write of 16-bits to FLASHADR.

3. Load FLASHCON with code to initiate a read (0x01). This is accomplished with an Indirect Write of 8-bits to FLASHCON.

4. Initiate the read by reading FLASHDAT. This is an Indirect Read of 0-bits (the DR scan consists of only the 2-bit read op-code). Note that this merely starts the FLASH read process.

5. Load FLASHCON with the code to poll FLBusy (0x00); This is an Indirect Write of 8-bits to FLASHCON.

6. Poll **FLBusy** until it goes low, indicating that the read has completed. This is an Indirect Read of 1-bit. The DR Scan for polling **FLBusy** is shown in Figure 14.

7. Read FLASHDAT. This is an Indirect Read of 10-bits (8 data bits, 1 **FLFail** bit, and 1

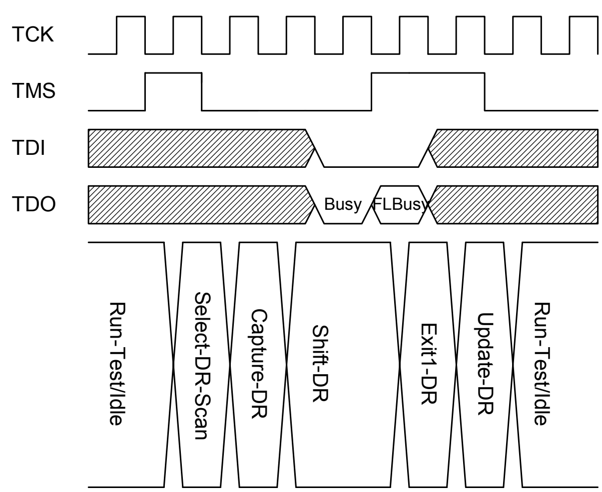

Figure 14. DR Scan Timing for Polling FLBusy

FLBusy bit). The DR Scan for reading FLASHDAT is shown in Figure 15.

If a series of consecutive bytes are to be read, the process can be restarted again at step (3) above, since FLASHADR is automatically incremented following a read or a write operation.

The **FLFail** bit is set to a ‘1’ if the read operation attempted to access a Read-locked sector.

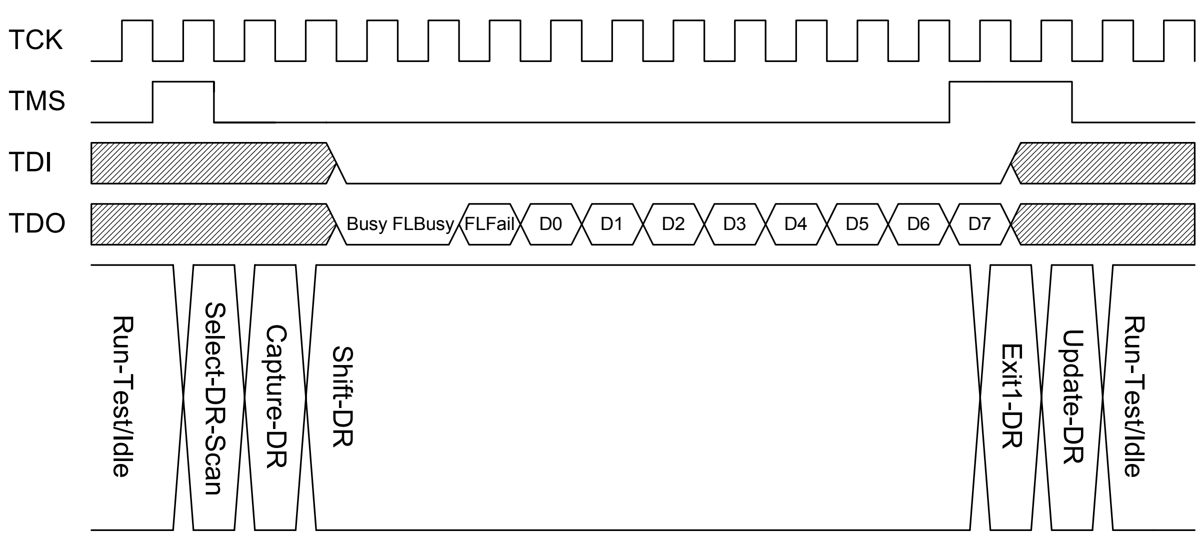

Figure 15. DR Scan Timing for FLASHDAT Read

Writing a FLASH Byte

Figure 16 shows a flow chart describing how to write a FLASH byte. The procedure is as follows:

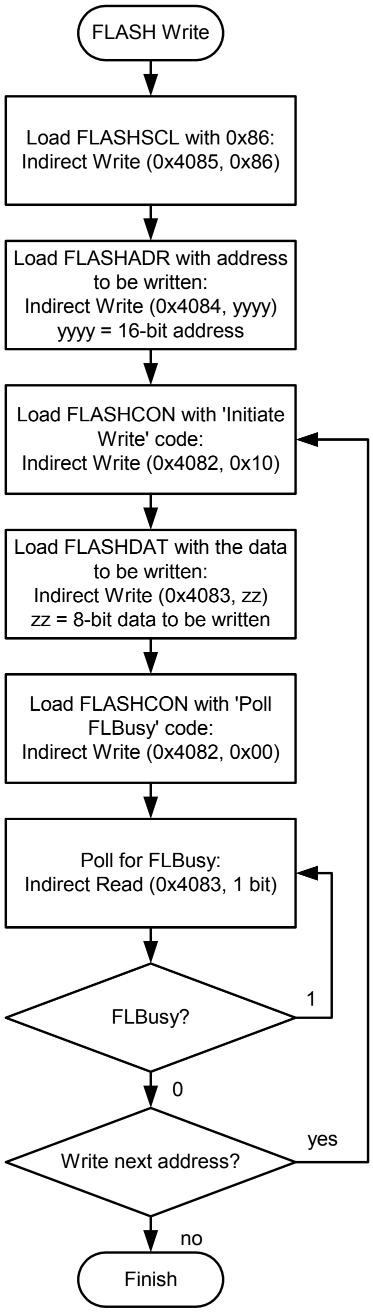

Figure 16. Flow Chart for Writing a FLASH Byte

1. Load FLASHSCL with 0x86, to set proper FLASH timing for using the internal 2 MHz system clock. This is accomplished by an Indirect Write of 0x86 to FLASHSCL.

2. Load FLASHADR with the 16-bit address to be written.

3. Load FLASHCON with the 'Initiate Write' opcode (0x10).

4. Load FLASHDAT with the data to be written. This is an 8-bit Indirect Write.

5. Load FLASHCON with the 'Poll **FLBusy**' opcode (0x00).

6. Poll **FLBusy**. This is accomplished by initiating 1-bit Indirect Reads on the FLASHDAT register.

If a series of consecutive bytes is to be written, the process can repeat, starting at step (3) above. FLASHADR is automatically incremented at the end of a read or a write operation.

The **FLFail** bit is set to a ‘1’ if the write operation attempted to write to a Write-locked sector.

Figure 17 shows the DR Scan timing for the 8-bit write to FLASHDAT, step (4) above.

Erasing a FLASH Page

The FLASH memory is organized as a series of 512-byte pages. The procedure for erasing a FLASH page is similar to writing a FLASH byte, except that the FLASHCON register needs to be set to 0x20, and FLASHDAT needs to be set to 0xA5. FLASHADR can be set to any address within the page to be erased. If FLASHADR is set to either of the Lock Byte addresses (0x7dfe or 0x7dff on 'F0xx devices and 0x1dfe or 0x1dff on the 'F2xx devices), then the erase operation initiates an erase of the entire FLASH memory.

Unlike read and write operations, FLASHADR is not automatically incremented at the end of an

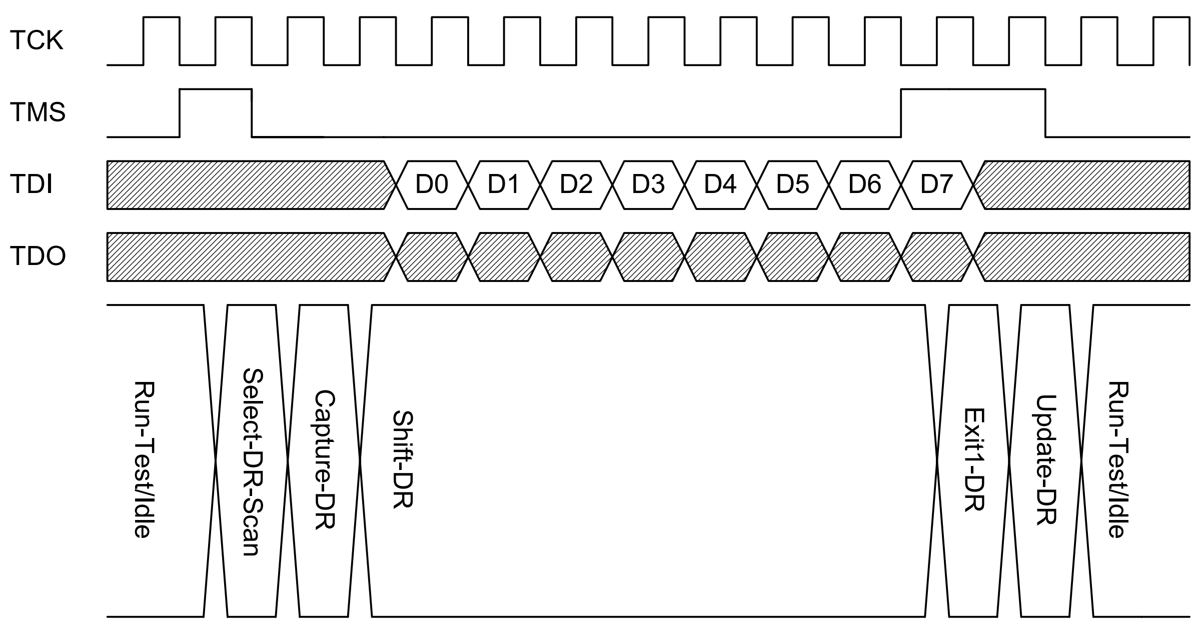

Figure 17. DR Scan Timing for a FLASHDAT Write

erase operation. Figure 18 shows a flow chart for the FLASH page erase procedure.

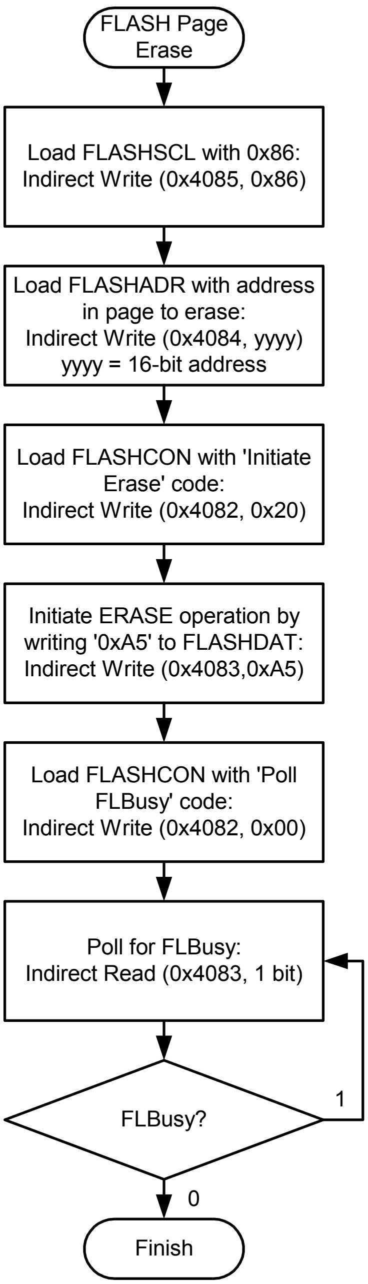

Figure 18. Flow Chart for Erasing a FLASH Page

Programming a Device in a JTAG Chain

If the C8051 device participates in a boundary scan chain with other devices or the JTAG ports of multiple C8051 devices are connected as shown in Figure 19, the device can be isolated and programmed using methods discussed in this note. A software example of programming the FLASH of a device in a JTAG chain is included at the end of this note.

Discovering an Unknown JTAG Chain

The purpose of the discovery process is to collect information about the devices connected in the chain. The discovery process assumes that all instruction registers have a '1' in the LSB and '0's in all other bit positions. This is true for all Silicon Labs devices, but may not be true for all JTAG devices. Also, upon reset, the optional 32-bit IDCODE register is selected by default. If the device does not have an IDCODE register, the 1-bit BYPASS register is selected instead. In the software example, the discovery process uses these assumptions to record information about the devices connected in the chain.

The discovery process is divided into two parts, an Instruction Register (IR) scan to determine the number of devices in the chain and the length of each device's Instruction Register, and a Data Register (DR) scan to collect each device's identification number. If a device does not support the IDCODE instruction, then it is assigned an ID of 0x00000000.

The Instruction Register discovery process begins with a JTAG_Reset operation. During the following IR_Scan operation, ones are shifted into the TDI pin on the last device in the chain (Device #2 in Figure 19, for example). The IR discovery process ends when a '11' pattern is received from the TDO pin of the first device in the JTAG chain (Device #2 in Figure 19). An input of '10' signifies that a new device has been encountered. Figure 20 shows the state machine used for analyzing the inputs in a discovery IR scan.

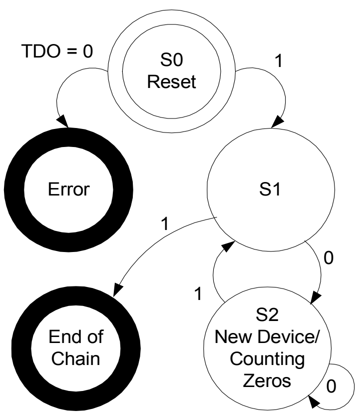

Figure 20. IR Discovery State Machine

After the IR scan is complete and the JTAG state machine is reset, the discovery process issues a DR scan to read and store the IDs of the devices for

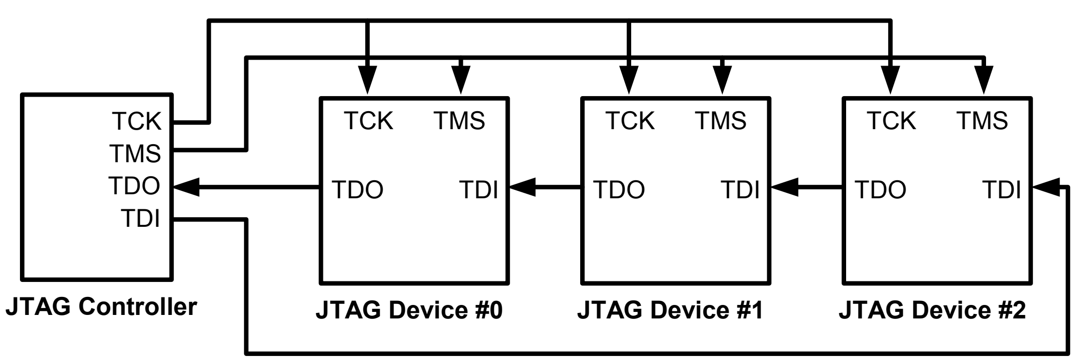

Figure 19. Typical JTAG Chain Connection

future reference. From the JTAG specification, the LSB returned on a DR scan will be a '1' if the device supports the IDCODE instruction and a '0' if the device is instead in BYPASS mode. Figure 21 shows how the DR scan determines each device's identification number.

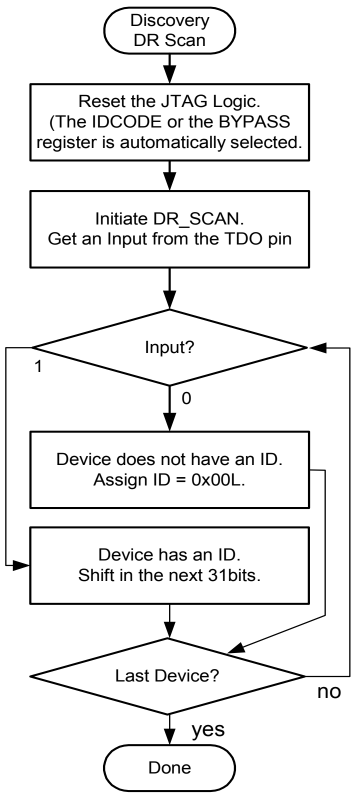

Figure 21. Flow Chart for Discovering the Device IDs

IR and DR Scans in JTAG Chains

Each Instruction Register scan operation is configured to place all devices other than the device to be programmed in BYPASS mode. This is accomplished by shifting '1's into the Instruction Registers of all devices before and after the isolated device, as shown in Figure 22.

Data Register scan operations pad one bit for each device before the device to be programmed and one bit for each device after the device to be programmed to account for the BYPASS registers of these devices.

1. IR Scan operations are prefixed with m '1's and post-fixed with n '1's, where m is the number of instruction register bits before the device to be programmed and n is the number of instruction register bits after the device to be programmed, as shown in Figure 22.

2. DR Scan operations are prefixed with x '0's and post-fixed with y '0's where x is the number of JTAG devices before the device to be programmed and y is the number of JTAG devices in the chain after the device to be programmed.

Isolating a Device

To be able to program a device in a chain, the device must be isolated. An isolated device is the only one not in BYPASS mode. This allows only one device to be accessed at a time. There are four variables that the IR scan and DR scan operations

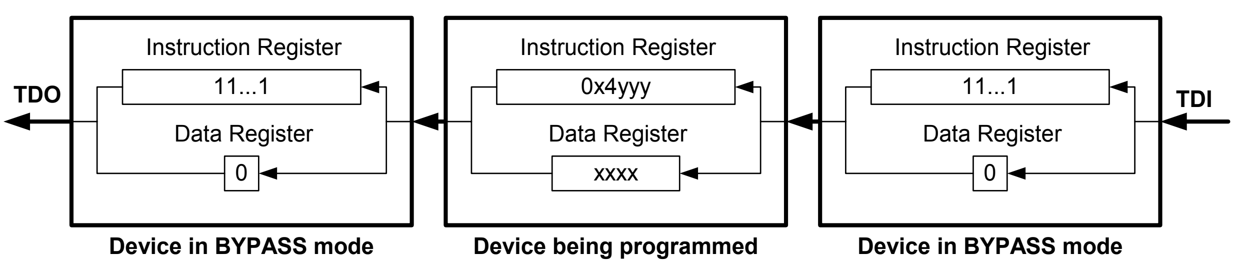

Figure 22. Isolating a C8051 Device to be Programmed

use to determine how many ‘1’s or ‘0’s to pad with when issuing a scan, as follows:

Table 12. Variables Required when Isolating a Device in a JTAG Chain.

For the IR_Scan operations:

number of IR bits before the isolated device
number of IR bits after the isolated device

For the DR_Scan operations:

number of devices before the isolated device
number of devices after the isolated device

In the software example, the JTAG_Isolate() procedure accepts the index of the device to be isolated and sets these variables accordingly. After the procedure is called, all the following IR and DR scans are performed on the device specified by the index. To execute a scan operation on another device, the JTAG_Isolate() procedure must be called with a new index prior to issuing the scan. If there is only one device in the chain, then neither the JTAG_Isolate() nor the JTAG_Discover() procedures need to be called prior to issuing a scan.

FLASH Operations

FLASH operations for a JTAG chain are the same as for a single device except that the device under test must be isolated before calling the FLASH Operations.

Software Examples For the ‘F00x, ‘F01x, and ‘F2xx Series

Programming a Single JTAG Device

```
//------------------------------------------------------------------------------------
// JTAG_FLASH.c
//------------------------------------------------------------------------------------
// This program contains some primitive routines which read, write, and erase the FLASH
// through the JTAG port on a C8051Fxxx device under test (DUT).  The JTAG pins on the
// DUT are connected to port pins on the C8051F000 master device.
//
// Target device: C8051F000, C8051F010
//
// Tool chain: KEIL Eval ‘c’
//

//------------------------------------------------------------------------------------
// Includes
//------------------------------------------------------------------------------------
#include <c8051f000.h>                          // SFR declarations

//------------------------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------------------------
sbit   LED = P1^6;                              // green LED: ‘1’ = ON; ‘0’ = OFF

// GPIO pins connecting to JTAG pins on device to be programmed (DUT)
sbit   TCK = P3^7;                              // JTAG Test Clock
sbit   TMS = P3^6;                              // JTAG Mode Select
sbit   TDI = P3^5;                              // JTAG Data Input
sbit   TDO = P3^4;                              // JTAG Data Output

#define   TRUE 1
#define   FALSE 0

// JTAG Instruction Register Addresses
#define   INST_LENGTH 16                        // number of bits in the
                                                // Instruction Register
#define   BYPASS      0xffff
#define   EXTEST      0x0000
#define   SAMPLE      0x0002

#define   RESET       0x2fff                    // System RESET Instruction

#define   IDCODE      0x1004                    // IDCODE Instruction address/HALT
#define   IDCODE_LEN  32                        // number of bits in the ID code

#define   FLASHCON    0x4082                    // FLASH Control Instruction address
#define   FLCN_LEN    8                         // number of bits in FLASHCON

#define   FLASHDAT    0x4083                    // FLASH Data Instruction address
#define   FLD_RDLEN   10                        // number of bits in an FLASHDAT read
#define   FLD_WRLEN   8                         // number of bits in an FLASHDAT write

#define   FLASHADR    0x4084                    // FLASH Address Instruction address
#define   FLA_LEN     16                        // number of bits in FLASHADR
```

```
#define   FLASHSCL    0x4085                 // FLASH Scale Instruction address
#define   FLSC_LEN    8                      // number of bits in FLASHSCL

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void init (void);
void JTAG_StrobeTCK (void);
void JTAG_Reset (void);
unsigned int JTAG_IR_Scan (unsigned int instruction, int num_bits);
unsigned long JTAG_DR_Scan (unsigned long dat, int num_bits);
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits);
unsigned long JTAG_IRead (unsigned int ireg, int num_bits);
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat);
int FLASH_ByteWrite (unsigned int addr, unsigned char dat);
int FLASH_PageErase (unsigned int addr);

//-----------------------------------------------------------------------------
// MAIN Routine

void main (void) {

   unsigned long id;
   unsigned char dest;
   int pass;

   id = 0x12345678L;

   init ();                                   // initialize ports

   JTAG_Reset ();                             // Reset the JTAG state machine on DUT

   JTAG_IR_Scan (RESET, INST_LENGTH);         // Reset the DUT

   JTAG_IR_Scan (IDCODE, INST_LENGTH);        // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);      // read the IDCODE
                                              // IDCODE should = 0x10000243 for
                                              // C8051F000 rev D device

   // here we erase the FLASH page 0x1000 - 0x11ff, read 0x1000 (it's an 0xff),
   // write a 0x66 to 0x1000, and read 0x1000 again (it's changed to an 0x66).
   while (1) {
      pass = FLASH_PageErase (0x7c00);        // erase page prior to writing...
      while (!pass);                          // handle Write Lock condition

      dest = 0x5a;                            // set test variable to non-0xff value

      pass = FLASH_ByteRead (0x7c00, &dest);  // dest should return 0xff
      while (!pass);                          // handle Read Lock condition

      dest = 0x66;
      pass = FLASH_ByteWrite (0x7c00, dest);  // store 0x66 at 0x1000
      while (!pass);                          // handle Read Lock condition

      pass = FLASH_ByteRead (0x7c00, &dest);  // dest should return 0x66
      while (!pass);                          // handle Read Lock condition

      pass = FLASH_PageErase (0x7c00);
```

```
      while (!pass);

      pass = FLASH_ByteRead (0x7c00, &dest);
      while (!pass);
   }
}

//-----------------------------------------------------------------------------
// Functions and Procedures
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// init
//-----------------------------------------------------------------------------
// This routine disables the watchdog timer and initializes the GPIO pins
//
void init (void) {

   WDTCN = 0xde;                          // disable watchdog timer
   WDTCN = 0xad;

   XBR2 |= 0x40;                          // enable crossbar
   PRT1CF |= 0x40;                        // enable P1.6 (LED) as a push-pull output
   PRT3CF |= 0xe0;                        // make P3.7-5 push-pull outputs
   P3 &= 0x1f;                            // TCK, TMS, and TDI all low

}

//-----------------------------------------------------------------------------
// JTAG_StrobeTCK
//-----------------------------------------------------------------------------
// This routine strobes the TCK pin (brings high then back low again)
// on the target system.
//
void JTAG_StrobeTCK (void) {

   TCK = 1;
   TCK = 0;
}

//-----------------------------------------------------------------------------
// JTAG_Reset
//-----------------------------------------------------------------------------
// This routine places the JTAG state machine on the target system in
// the Test Logic Reset state by strobing TCK 5 times while leaving
// TMS high.  Leaves the JTAG state machine in the Run_Test/Idle state.
//
void JTAG_Reset (void) {

   TMS = 1;

   JTAG_StrobeTCK ();                     // move to Test Logic Reset state
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();

   TMS = 0;
```

```
   JTAG_StrobeTCK ();                         // move to Run_Test/Idle state
}

//-----------------------------------------------------------------------------
// JTAG_IR_Scan
//-----------------------------------------------------------------------------
// This routine loads the supplied <instruction> of <num_bits> length into the JTAG
// Instruction Register on the target system.  Leaves in the Run_Test/Idle state.
// The return value is the n-bit value read from the IR.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
//
unsigned int JTAG_IR_Scan (unsigned int instruction, int num_bits) {

   unsigned int retval;                       // JTAG instruction read
   int   i;                                   // JTAG IR bit counter

   retval = 0x0;

   TMS = 1;
   JTAG_StrobeTCK ();                         // move to SelectDR
   TMS = 1;
   JTAG_StrobeTCK ();                         // move to SelectIR
   TMS = 0;
   JTAG_StrobeTCK ();                         // move to Capture_IR
   TMS = 0;
   JTAG_StrobeTCK ();                         // move to Shift_IR state

   for (i=0; i < num_bits; i++) {

      TDI = (instruction & 0x01);             // shift IR, LSB-first
      instruction = instruction >> 1;

      retval = retval >> 1;
      if (TDO) {
         retval |= (0x01 << (num_bits - 1));
      }

      if (i == (num_bits - 1)) {
         TMS = 1;                             // move to Exit1_IR state
      }

      JTAG_StrobeTCK();
   }

   TMS = 1;
   JTAG_StrobeTCK ();                         // move to Update_IR
   TMS = 0;
   JTAG_StrobeTCK ();                         // move to RTI state

   return retval;
}

//-----------------------------------------------------------------------------
// JTAG_DR_Scan
//-----------------------------------------------------------------------------
// This routine shifts <num_bits> of <data> into the Data Register, and returns
// up to 32-bits of data read from the Data Register.
// Leaves in the Run_Test/Idle state.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
```

```
//
unsigned long JTAG_DR_Scan (unsigned long dat, int num_bits) {

   unsigned long   retval;                 // JTAG return value
   int   i;                                // JTAG DR bit counter

   retval = 0x0L;

   TMS = 1;
   JTAG_StrobeTCK ();                      // move to SelectDR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Capture_DR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Shift_DR state

   for (i=0; i < num_bits; i++) {

      TDI = (dat & 0x01);                  // shift DR, LSB-first
      dat = dat >> 1;

      retval = retval >> 1;
      if (TDO) {
         retval |= (0x01L << (num_bits - 1));
      }

      if ( i == (num_bits - 1)) {
         TMS = 1;                          // move to Exit1_DR state
      }

      JTAG_StrobeTCK();
   }
   TMS = 1;
   JTAG_StrobeTCK ();                      // move to Update_DR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to RTI state

   return retval;
}

//------------------------------------------------------------------------------------
// JTAG_IWrite
//------------------------------------------------------------------------------------
// This routine performs an indirect write to register <ireg>, containing <dat>, of
// <num_bits> in length.  It follows the write operation with a polling operation, and
// returns when the operation is completed.  Note: the polling implemented here refers
// to the JTAG register write operation being completed, NOT the FLASH write operation.
// Polling for the FLASH write operation is handled at a higher level
// Examples of valid indirect registers are:
//  FLCN - FLASH Control
//  FLSC - FLASH Scale
//  FLA - FLASH Address
//  FLD - FLASH Data
// Leaves in the Run_Test/Idle state.
//
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits) {

   int done;                               // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);       // load IR with <ireg>
```

```
   dat |= (0x03L << num_bits);            // append 'WRITE' opcode to data

   // load DR with <dat>
   JTAG_DR_Scan (dat, num_bits + 2);      // initiate the JTAG write

   // load DR with '0', and check for BUSY bit to go to '0'.
   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);
}

//-------------------------------------------------------------------------------------
// JTAG_IRead
//-------------------------------------------------------------------------------------
// This routine performs an indirect read of register <ireg>, of <num_bits> in length.
// It follows the read operation with a polling operation, and returns when the
// operation is completed.  Note: the polling implemented here refers to the JTAG
// register read operation being completed, NOT the FLASH read operation.
// Polling for the FLASH read operation is handled at a higher level.
// Examples of valid indirect registers are:
//  FLCN - FLASH Control
//  FLSC - FLASH Scale
//  FLA - FLASH Address
//  FLD - FLASH Data
// Leaves in the Run_Test/Idle state.
//
unsigned long JTAG_IRead (unsigned int ireg, int num_bits) {

   unsigned long retval;                  // value returned from READ operation
   int done;                              // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);      // load IR with <ireg>

   // load DR with read opcode (0x02)

   JTAG_DR_Scan (0x02L, 2);               // initiate the JTAG read

   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);

   retval = JTAG_DR_Scan (0x0L, num_bits + 1);  // allow poll operation to
                                                // read remainder of the bits
   retval = retval >> 1;                        // shift JTAG_BUSY bit off the end

   return retval;
}

//-------------------------------------------------------------------------------------
// FLASH_ByteRead
//-------------------------------------------------------------------------------------
// This routine reads the byte at <addr> and stores it at the address pointed to by
// <pdat>.
// Returns TRUE if the operation was successful; FALSE otherwise (page read-protected).
//
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat)
{
   unsigned long testval;                      // holds result of FLASHDAT read
```

```
   int done;                                      // TRUE/FALSE flag
   int retval;                                    // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);       // set FLASHSCL based on SYSCLK
                                                  //  frequency (2MHz = 0x86)

   // set FLASHADR to address to read from
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x02L, FLCN_LEN);       // set FLASHCON for FLASH Read
                                                  //  operation (0x02)

   JTAG_IRead (FLASHDAT, FLD_RDLEN);              // initiate the read operation

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);        // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));         // poll for FLBUSY to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, FLD_RDLEN);    // read the resulting data

   retval = (testval & 0x02) ? FALSE: TRUE;       // FLFail is next to LSB

   testval = testval >> 2;                        // shift data.0 into LSB position

   *pdat = (unsigned char) testval;               // place data in return location

   return retval;                                 // return FLASH Pass/Fail
}

//------------------------------------------------------------------------------------
// FLASH_ByteWrite
//------------------------------------------------------------------------------------
// This routine writes the data <dat> to FLASH at the address <addr>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
// write-protected).
//
int FLASH_ByteWrite (unsigned int addr, unsigned char dat)
{
   unsigned long testval;                         // holds result of FLASHDAT read
   int done;                                      // TRUE/FALSE flag
   int retval;                                    // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);       // set FLASHSCL based on SYSCLK
                                                  //  frequency (2MHz = 0x86)

   // set FLASHADR to address to write to
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x10L, FLCN_LEN);       // set FLASHCON for FLASH Write
                                                  //  operation (0x10)

   // initiate the write operation
   JTAG_IWrite (FLASHDAT, (unsigned long) dat, FLD_WRLEN);

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);        // set FLASHCON for 'poll' operation

   do {
```

```
      done = !(JTAG_IRead (FLASHDAT, 1));      // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);         // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;    // FLFail is next to LSB

   return retval;                              // return FLASH Pass/Fail
}

//------------------------------------------------------------------------------------
// FLASH_PageErase
//------------------------------------------------------------------------------------
// This routine performs an erase of the page in which <addr> is contained.
// This routine assumes that no FLASH operations are currently in progress.
// This routine exits with no FLASH operations currently in progress.
// Returns TRUE if the operation was successful; FALSE otherwise (page protected).
//
int FLASH_PageErase (unsigned int addr)
{
   unsigned long testval;                      // holds result of FLASHDAT read
   int done;                                   // TRUE/FALSE flag
   int retval;                                 // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);    // set FLASHSCL based on SYSCLK
                                               //  frequency (2MHz = 0x86)

   // set FLASHADR to address within page to erase
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x20L, FLCN_LEN);    // set FLASHCON for FLASH Erase
                                               //  operation (0x20)

   JTAG_IWrite (FLASHDAT, 0xa5L, FLD_WRLEN);   // set FLASHDAT to 0xa5 to initiate
                                               //  erase procedure

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);     // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));      // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);         // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;    // FLFail is next to LSB

   // set return value based on FLFail bit
   return retval;                              // return FLASH Pass/Fail
}
```

Programming Multiple JTAG Devices in a Chain

```
//************************************************************************************
// JTAG_Chain.c
//------------------------------------------------------------------------------------
// This program contains some primitive routines which gather information through the
// JTAG port on multiple JTAG compatible devices under test (DUT) connected in a
// chain. The TCK & TMS JTAG pins on the DUT are connected in parallel to port pins on
// the C8051F00x, C8051F01x master device and the TDI & TDO pins are connected in
// series.
//
// **NOTE: The first device in the chain (device 0) is the one whose TDO pin is
//         connected to the TDO pin of the master device.
//
// Target device: C8051F00x,C8051F01x
//
// Tool chain: KEIL Eval 'c'
//************************************************************************************

//------------------------------------------------------------------------------------
// Includes
//------------------------------------------------------------------------------------

#include <c8051f000.h>                      // SFR declarations

//------------------------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------------------------

#define MAX_NUM_DEVICES_IN_CHAIN 10

#define SYSCLK 2000000                      // SYSCLK frequency in Hz

sbit   LED = P1^6;                          // green LED: '1' = ON; '0' = OFF

sbit   TCK = P3^7;      // JTAG Test Clock -- Connected to TCK pin on all devices.
sbit   TMS = P3^6;      // JTAG Mode Select -- Connected to TMS pin on all devices.
sbit   TDI = P3^5;      // JTAG Data Input(output of master) -- Connected to the
                        //      TDI pin of device n.
sbit   TDO = P3^4;      // JTAG Data Output (input to master)-- Connected to the
                        //      TDO pin of device 0.

#define   TRUE 1
#define   FALSE 0

// JTAG Instruction Register Addresses
#define   INST_LENGTH 16                   // number of bits in the C8051Fxxx
#define   BYPASS      0xffff               // Instruction Register
#define   EXTEST      0x0000
#define   SAMPLE      0x0002

#define   RESET       0x2fff               // System RESET Instruction

#define   IDCODE      0x1004               // IDCODE Instruction address/HALT
#define   IDCODE_LEN  32                   // number of bits in the ID code

#define   FLASHCON    0x4082               // FLASH Control Instruction address
#define   FLCN_LEN    8                    // number of bits in FLASHCON
```

```
#define   FLASHDAT    0x4083                 // FLASH Data Instruction address
#define   FLD_RDLEN   10                     // number of bits in an FLASHDAT read
#define   FLD_WRLEN   8                      // number of bits in an FLASHDAT write

#define   FLASHADR    0x4084                 // FLASH Address Instruction address
#define   FLA_LEN     16                     // number of bits in FLASHADR

#define   FLASHSCL    0x4085                 // FLASH Scale Instruction address
#define   FLSC_LEN    8                      // number of bits in FLASHSCL
//-----------------------------------------------------------------------------------
// Global Variable DECLARATIONS
//-----------------------------------------------------------------------------------

// The addresses of the following variables are explicitly defined for viewing
// purposes. If the width of the external memory window is 5 bytes, then each
// device will take up exactly one row starting from the second row.
char xdata num_devices  _at_ 0x0000;

char xdata num_devices_before _at_ 0x0001; // #devices before and after the isolated
char xdata num_devices_after  _at_ 0x0002; // device
char xdata num_IR_bits_before _at_ 0x0003; // #instruction register bits before and
char xdata num_IR_bits_after  _at_ 0x0004; // after the isolated device


typedef struct JTAG_Information {          // Discovery information
      unsigned char IR_length;             // Instruction register length
      unsigned long id;                    // Identification code for each device
} JTAG_Information;
                                           // Array: one entry per device in the
                                           // JTAG chain
JTAG_Information xdata JTAG_info[MAX_NUM_DEVICES_IN_CHAIN];

//-----------------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------------

void init (void);
void JTAG_StrobeTCK (void);
void JTAG_Reset (void);

void Blink_Led(void);

void JTAG_Discover(void);
void JTAG_Discover_IR(void);
void JTAG_Discover_DR(void);
void JTAG_Isolate(char index);

unsigned long JTAG_IR_Scan (unsigned long instruction, char num_bits) ;
unsigned long JTAG_DR_Scan (unsigned long dat, char num_bits);

void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits);
unsigned long JTAG_IRead (unsigned int ireg, int num_bits);
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat);
int FLASH_ByteWrite (unsigned int addr, unsigned char dat);
int FLASH_PageErase (unsigned int addr);

//-----------------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------------
```

```
void main (void)
{

   long xdata id;
   unsigned char dest;
   int pass;
   int address;
   char device = 0;


   init ();                                 // initialize ports

   LED = 1;                                 // turn on the LED

   JTAG_Discover();                         // IDCODE should = 0x10000243 for
                                            // C8051F000 rev D device
   JTAG_Isolate(0);                         // isolate device 0
   JTAG_IR_Scan (IDCODE, INST_LENGTH);      // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);    // get the ID Code of the isolated device

   JTAG_Isolate(1);
   JTAG_IR_Scan (IDCODE, INST_LENGTH);      // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);    // get the ID Code of the isolated device

   JTAG_Isolate(2);
   JTAG_IR_Scan (IDCODE, INST_LENGTH);      // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);    // get the ID Code of the isolated device


   // Here we perform 2 tests on each device.  These 2 tests take approximately
   // 43 seconds for each device with SYSCLK at 2 Mhz and approximatly 6 seconds
   // running at 16 Mhz.

   for(device = 0; device < num_devices; device++) {

      JTAG_Isolate(device);

      //TEST 1 -- ERASE A FLASH PAGE
      pass = FLASH_PageErase (0x1000);       // erase page prior to writing
      while (!pass);                         // handle Write Lock condition


      //Verify that locations 0x1000 - 0x11FF are 0xFF
      for(address = 0x1000; address < 0x1200; address++){
         pass = FLASH_ByteRead (address, &dest);   // dest should return 0xff
         if(!pass || dest != 0xFF) Blink_Led();
      }

      //TEST 2 -- WRITE A PATTERN TO FLASH PAGE

      for(address = 0x1000; address < 0x1200; address++){
         dest = address & 0x00FF;               // strip away upper 8 bits
         pass = FLASH_ByteWrite (address, dest);// store LSByte of address at address
         while (!pass);                         // handle Read Lock condition
      }

      dest = 0x12;                              // set test variable to non-0xff value

      //Verify that locations 0x1000 - 0x11FF are following the pattern
```

```
      for(address = 0x1000; address < 0x1200; address++){
         pass = FLASH_ByteRead (address, &dest);
         if(!pass || dest != (address & 0x00FF)) Blink_Led();
      }
   }


   LED = 0;                                          // turn off the led,
                                                     // program executed correctly
   while(1);
}

//*************************************************************************************
// Function and Procedure DEFINITIONS
//*************************************************************************************

//-------------------------------------------------------------------------------------
// Blink_Led
//-------------------------------------------------------------------------------------
// This routine blinks the Green LED forever to indicate an error.
//
void Blink_Led(void)
{
   int i;                                         // millisecond counter
   int ms = 200;                                  // stay in each state for ms milliseconds

   TCON  &= ~0x30;                                // STOP Timer0 and clear overflow flag
   TMOD  &= ~0x0F;                                // configure Timer0 to 16-bit mode
   TMOD  |=  0x01;
   CKCON |=  0x08;                                // Timer0 counts SYSCLKs


   while (1){

   LED = ~LED;

      for (i = 0; i < ms; i++) {                  // count milliseconds
         TR0 = 0;                                 // STOP Timer0
         TH0 = (-SYSCLK/1000) >> 8;               // SET Timer0 to overflow in 1ms
         TL0 = -SYSCLK/1000;
         TR0 = 1;                                 // START Timer0

         while(TF0 == 0);                         // wait for overflow

         TF0 = 0;                                 // clear overflow indicator
      }
   }
}

//-------------------------------------------------------------------------------------
// init
//-------------------------------------------------------------------------------------
// This routine disables the watchdog timer and initializes the GPIO pins
//
void init (void)
{

   WDTCN = 0xde;                                 // disable watchdog timer
   WDTCN = 0xad;
```

```
   XBR2   |=  0x40;                        // enable crossbar
   PRT1CF |=  0x40;                        // enable P1.6 (LED) as a push-pull output
   PRT3CF |=  0xe0;                        // make P3.7-5 push-pull outputs
   P3     &= ~0xE0;                        // set TCK, TMS, and TDI all low

   num_devices = 1;                        // The default number of devices is one.
                                           // JTAG_Discover() does not have to be
                                           // called if only one device is connected.

   num_devices_before = 0;                 // Initializing these variables to zero
   num_devices_after = 0;                  // allows calling the JTAG_IR_Scan() and
   num_IR_bits_before = 0;                 // the JTAG_DR_Scan() without first
   num_IR_bits_after = 0;                  // calling JTAG_Isolate() when there is
                                           // only one device in the chain.
}

//------------------------------------------------------------------------------------
// JTAG_StrobeTCK
//------------------------------------------------------------------------------------
// This routine strobes the TCK pin (brings high then back low again)
// on the target system.
//
void JTAG_StrobeTCK (void)
{

   TCK = 1;
   TCK = 0;
}

//------------------------------------------------------------------------------------
// JTAG_Reset
//------------------------------------------------------------------------------------
// This routine places the JTAG state machine on the target system in
// the Test Logic Reset state by strobing TCK 5 times while leaving
// TMS high.  Leaves the JTAG state machine in the Run_Test/Idle state.
//
void JTAG_Reset (void)
{

   TMS = 1;

   JTAG_StrobeTCK ();                      // move to Test Logic Reset state
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();

   TMS = 0;

   JTAG_StrobeTCK ();                      // move to Run_Test/Idle state
}

//------------------------------------------------------------------------------------
// JTAG_Discover
//------------------------------------------------------------------------------------
// This routine sequentially queries a chain of JTAG devices and accomplishes the
// following three tasks.
//    For the global struct array <JTAG_info>
```

```
//      -- fills in the length of each device's instruction register
//      -- fills in each device's IDCODE.
//    For the global variable <num_devices>
//      -- updates it with the number of JTAG devices connected in the chain.
//
void JTAG_Discover(void)
{

   JTAG_Discover_IR();

   // At this point we know num_devices(a global variable) and we know the
   // length of each device's IR given in the variable JTAG_info[].IR_length

   JTAG_Discover_DR();                       // Read and assign the ID for each
                                             // device

} //end discover

//-------------------------------------------------------------------------------------
// JTAG_Discover_IR
//-------------------------------------------------------------------------------------
// This routine fills a structure with the length of each device's instruction
// register. It also updates the global variable <num_devices> with the number of
// JTAG devices connected in the chain.
//
// BACKGROUND: When an IRSCAN is issued, a JTAG device must return a 1 as the LSB
//             and zeros in all the other bits.  We shift in all ones so when we
//             encounter two ones in a row, we know we are past the end of the chain.
//             A state machine is implemented in this routine to keep track of
//             inputs received.
//
// STATE DEFINITONS:
//             0 - NO INPUTS -- at beginning of chain
//          1 - INPUT SEQUENCE: 1 -- could be at a new device or at chain end
//             2 - INPUT SEQUENCE: 100..0 -- counting zeros
//
//
void JTAG_Discover_IR(void)
{

   char state = 0;                          // beginning of chain

   char num_zeros = 0;                      // number of zeros following a one in
                                            // an IR_SCAN. num_zeros + 1 = IR_length

   char current_device_index = -1;          // current_device_index + 1 = num_devices
                                            // (on the last iteration)
   bit done = FALSE;                        // TRUE when end of chain is reached

   JTAG_Reset();                            // RESET and move to Run_Test/Idle

   // advance to Shift_IR State
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to SelectDR
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to SelectIR
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to Capture_IR
   TMS = 0;
```

```
JTAG_StrobeTCK ();                          // move to Shift_IR state and get the
                                            // the first input

TDI = 1;                                    // STATE is initially 0
                                            // shift in all ones
// for each device
do{

   if(TDO != 1){                            // Error if the first input is not one.
      Blink_Led();                          // Could mean bad connections or
   }                                        // non-compliant devices.



   state = 1;                               // received a 1, could be at a new
                                            // device or at the end of the chain

   num_zeros = 0;                           // initialize for the zero counting loop

   // for the number of zeros in each device’s IR
   do {

      JTAG_StrobeTCK();                     // get the next bit.

      switch(state){

         case 1: if(TDO == 0){              // found new device(10)
                    current_device_index++;
                    num_zeros++;
                    state = 2;
                 } else {
                    done = TRUE;            // at end of chain (11)
                 }
                 break;

         case 2: if(TDO == 0){
                   num_zeros++;             // counting zeros (10..0)
                } else {
                   state = 1;               // past end of current device (10..01)
                }
                break;

         default: Blink_Led();              // an error has occurred

      } // end switch


   } while ((state != 1) && (!done));       // while the input is not one,
                                            // count zeros until we get a one.

   if (!done) {                             // if we are not past the last device

      JTAG_info[current_device_index].IR_length = num_zeros + 1;
   }

} while (!done);                            //while we are not past the last device

num_devices = current_device_index + 1;
```

```
   // navigate the JTAG State Machine back to RTI state.
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to Exit1_IR state
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to Update_IR state
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to Run_Test/Idle state

}

//------------------------------------------------------------------------------------
// JTAG_Discover_DR
//------------------------------------------------------------------------------------
//GOAL: Obtain the ID code of each device(If it supports IDCODE), and fill in
//      the field JTAG_info[].id (32-bit).
//      Assign all zeros if device does not have an IDCODE.
//
//BACKGROUND: After JTAG State Machine Reset, the IDCODE is automatically selected
//            If a device does not have an IDCODE register, the BYPASS
//            register is selected instead.
//            On a DR_SCAN, each IDCODE register returns a 32-bit ID with LSB = 1
//            and each BYPASS register returns 1-bit = 0.

void JTAG_Discover_DR(void)
{

   char current_device_index = 0;

   unsigned char i;                       // loop counter


   JTAG_Reset ();                         // Reset the JTAG state machine on DUT
                                          // move to Run_Test/Idle

   // The IDCODE or the BYPASS Register is automatically selected.

   // Navigate to the Shift_DR state
   TMS = 1;
   JTAG_StrobeTCK ();                     // move to SelectDR
   TMS = 0;
   JTAG_StrobeTCK ();                     // move to Capture_DR

   TMS = 0;
   TDI = 1;                               // shift in all ones

   current_device_index = 0;

   while (current_device_index < num_devices) {

      JTAG_StrobeTCK ();                  // move to Shift_DR state and get input

      if (TDO == 0) {                     // Device does not have an IDCODE register

         JTAG_info[current_device_index].id = 0x00000000L;

      } else { // TDO == 1

         JTAG_info[current_device_index].id = 0x80000000L;
```

```
        for (i = 0; i < 31; i++){        // Get the next 31-bits of the device ID

           JTAG_StrobeTCK ();

           JTAG_info[current_device_index].id =
             JTAG_info[current_device_index].id >> 1;

           if (TDO) {
              JTAG_info[current_device_index].id |= 0x80000000L;
           }
        } // end for

     } // end if-else

     current_device_index++;

  } // end while

  //fill the rest of the entries with zeros
  for (; current_device_index < MAX_NUM_DEVICES_IN_CHAIN; current_device_index++) {

     JTAG_info[current_device_index].IR_length = 0;
     JTAG_info[current_device_index].id = 0x00000000L;
  }

  // Navigate JTAG State Machine back to RTI state
  TMS = 1;
  JTAG_StrobeTCK ();                     // move to Exit1_DR
  TMS = 1;
  JTAG_StrobeTCK ();                     // move to Update DR
  TMS = 0;
  JTAG_StrobeTCK ();                     // move to RTI

}

//------------------------------------------------------------------------------------
// JTAG_Isolate
//------------------------------------------------------------------------------------
// This routine updates 4 global variables.  JTAG_Discover() must be called prior to
// calling this routine in order to set up the data structure.
//
// VARIABLE DEFINITIONS
//    num_IR_bits_before -- number of instruction register bits before the isolated
//                          device
//    num_IR_bits_after  -- number of instruction register bits after the isolated
//                          device
//    num_devices_before -- number of devices before the isolated device
//    num_devices_after  -- number of device after the isolated device
//
void JTAG_Isolate(char index)
{

  unsigned char i;

  if ((index > (num_devices - 1)) || (index < 0) ) {
                                         // check if index is out of range
     Blink_Led();
  }
```

```
   num_devices_before = index;

   num_devices_after = num_devices - index - 1;

   num_IR_bits_before = 0;                // initializing for loop
   num_IR_bits_after = 0;

   for (i = 0; i < num_devices; i++) {

      if (i < index) {

         num_IR_bits_before += JTAG_info[i].IR_length;

      } else if (i > index) {

         num_IR_bits_after += JTAG_info[i].IR_length;
      }

      // last case -- equal, do nothing
   } // end for
} //end isolate


//-----------------------------------------------------------------------------------
// JTAG_IR_Scan
//-----------------------------------------------------------------------------------
// This routine loads the supplied <instruction> of <num_bits> length into the JTAG
// Instruction Register on the isolated device. It shifts the BYPASS opcode (all ones)
// into the Instruction Registers of the other devices in the chain.
//
// NOTE: JTAG_Discover() must be called before this function is called.
//
// NOTE: If more than one device is connected in the chain, JTAG_Isolate() must also
//       be called prior to calling this function.
//
// The return value is the n-bit value read from the IR.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
// Leaves JTAG in the Run_Test/Idle state.
//
unsigned long JTAG_IR_Scan (unsigned long instruction, char num_bits)
{

    unsigned long retval;                        // JTAG instruction read
    char i;                                      // JTAG IR bit counter

    retval = 0x0L;

    // navigate the JTAG State Machine in all devices to the Shift_IR state
    TMS = 1;
    JTAG_StrobeTCK ();                           // move to SelectDR
    TMS = 1;
    JTAG_StrobeTCK ();                           // move to SelectIR
    TMS = 0;
    JTAG_StrobeTCK ();                           // move to Capture_IR
    TMS = 0;
    JTAG_StrobeTCK ();                           // move to Shift_IR state

    TDI=1;
    for (i=0; i < num_IR_bits_before; i++) {
```

```
        JTAG_StrobeTCK();                          // fill the IR of the devices
                                                   // before the isolated device
    }                                              // with all ones, the BYPASS opcode


    for (i=0; i < num_bits; i++) {

        TDI = (instruction & 0x01);                // determine output
        instruction = instruction >> 1;

        retval = retval >> 1;
        if (TDO) {
            retval |= (0x01 << (num_bits - 1));
        }

        if ((i == (num_bits - 1)) && (num_IR_bits_after == 0)) {
            TMS = 1;                               // move to Exit1_IR state
        }

        JTAG_StrobeTCK();                          // move to Shift_IR state
                                                   // advance
    }

    TDI = 1;
    for (i=0; i < num_IR_bits_after; i++) {        // now process IR bits after the
                                                   // isolated device
        if (i == (num_IR_bits_after - 1)) {
            TMS = 1;                               // move to Exit1_IR state
        }


        JTAG_StrobeTCK();                          // fill the IR of the devices
                                                   // after the isolated device
    }                                              // with all ones, the BYPASS opcode.

    // navigate back to the RTI state

    TMS = 1;
    JTAG_StrobeTCK ();                             // move to Update_IR
    TMS = 0;
    JTAG_StrobeTCK ();                             // move to RTI state

    return retval;
}

//-----------------------------------------------------------------------------------------
// JTAG_DR_Scan
//-----------------------------------------------------------------------------------------
// This routine shifts <num_bits> of <data> into the Data Register of the isolated
// device in the chain, and returns up to 32-bits of data read from its Data Register.
//
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
// Leaves in the Run_Test/Idle state.
//
unsigned long JTAG_DR_Scan (unsigned long dat, char num_bits)
{

    unsigned long retval;                         // JTAG return value
```

```
char i;                                         // JTAG DR bit counter

retval = 0x0L;

// navigate the JTAG State Machine in all devices to the Shift_DR state
TMS = 1;
JTAG_StrobeTCK ();                              // move to SelectDR
TMS = 0;
JTAG_StrobeTCK ();                              // move to Capture_DR
TMS = 0;
JTAG_StrobeTCK ();                              // move to Shift_DR state


TDI = 0;
for (i=0; i < num_devices_before; i++) {

   JTAG_StrobeTCK();                            // fill the BYPASS Register
                                                // of the devices before the
}                                               // isolated device with zeros.



for (i=0; i < num_bits; i++) {

    TDI = (dat & 0x01);                         // determine the output
    dat = dat >> 1;

    retval = retval >> 1;
    if (TDO) {
        retval |= (0x01L << (num_bits - 1));
    }

    if ((i == (num_bits - 1)) && (num_devices_after == 0)) {
        TMS = 1;                                // move to Exit1_IR state
    }

    JTAG_StrobeTCK();
                                                //output and get input
}

TDI = 0;
for (i=0; i < num_devices_after; i++) {

    if (i == (num_devices_after - 1)) {
        TMS = 1;                                // move to Exit1_IR state
    }

                                                // move to Shift_DR state,
   JTAG_StrobeTCK();                            // fill the BYPASS Register
                                                // of the devices after the
}                                               // isolated device with zeros.

// navigate the JTAG State Machine in all devices to the RTI state
TMS = 1;
JTAG_StrobeTCK ();                              // move to Update_DR
TMS = 0;
JTAG_StrobeTCK ();                              // move to RTI state

return retval;                                  // retval is MSB aligned
```

```
}

//-----------------------------------------------------------------------------------
// JTAG_IWrite
//-----------------------------------------------------------------------------------
// This routine performs an indirect write to register <ireg>, containing <dat>, of
// <num_bits> in length.  It follows the write operation with a polling operation, and
// returns when the operation is completed.  Note: the polling implemented here refers
// to the JTAG register write operation being completed, NOT the FLASH write operation.
// Polling for the FLASH write operation is handled at a higher level
// Examples of valid indirect registers are:
//  FLASHCON - FLASH Control
//  FLASHSCL - FLASH Scale
//  FLASHADR - FLASH Address
//  FLASHDAT - FLASH Data
// Leaves in the Run_Test/Idle state.
//
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits)
{

   bit done;                            // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);      // load IR with <ireg>

   dat |= (0x03L << num_bits);            // append 'WRITE' opcode to data

   // load DR with <dat>
   JTAG_DR_Scan (dat, num_bits + 2);      // initiate the JTAG write

   // load DR with '0', and check for BUSY bit to go to '0'.
   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);
}

//-----------------------------------------------------------------------------------
// JTAG_IRead
//-----------------------------------------------------------------------------------
// This routine performs an indirect read of register <ireg>, of <num_bits> in length.
// It follows the read operation with a polling operation, and returns when the
// operation is completed.  Note: the polling implemented here refers to the JTAG
// register read operation being completed, NOT the FLASH read operation.
// Polling for the FLASH read operation is handled at a higher level.
// Examples of valid indirect registers are:
//  FLASHCON - FLASH Control
//  FLASHSCL - FLASH Scale
//  FLASHADR - FLASH Address
//  FLASHDAT - FLASH Data
// Leaves JTAG in the Run_Test/Idle state.
//
unsigned long JTAG_IRead (unsigned int ireg, int num_bits) {

   unsigned long retval;                // value returned from READ operation
   bit done;                            // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);      // load IR with <ireg>

   // load DR with read opcode (0x02)
   JTAG_DR_Scan (0x02L, 2);               // initiate the JTAG read
```

```
   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);

   retval = JTAG_DR_Scan (0x0L, num_bits + 1);  // allow poll operation to
                                                // read remainder of the bits
   retval = retval >> 1;                  // shift JTAG_BUSY bit off the end

   return retval;
}

//------------------------------------------------------------------------------------
// FLASH_ByteRead
//------------------------------------------------------------------------------------
// This routine reads the byte at <addr> and stores it at the address pointed to by
// <pdat>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
// read-protected).
//
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   bit done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address to read from
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x02L, FLCN_LEN);     // set FLASHCON for FLASH Read
                                                // operation (0x02)

   JTAG_IRead (FLASHDAT, FLD_RDLEN);            // initiate the read operation

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for ‘poll’ operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBUSY to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, FLD_RDLEN);  // read the resulting data

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   testval = testval >> 2;                      // shift data.0 into LSB position

   *pdat = (unsigned char) testval;             // place data in return location

   return retval;                               // return FLASH Pass/Fail
}

//------------------------------------------------------------------------------------
// FLASH_ByteWrite
//------------------------------------------------------------------------------------
// This routine writes the data <dat> to FLASH at the address <addr>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
```

```
// write-protected).
//
int FLASH_ByteWrite (unsigned int addr, unsigned char dat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   int done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address to write to
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x10L, FLCN_LEN);     // set FLASHCON for FLASH Write
                                                // operation (0x10)

   // initiate the write operation
   JTAG_IWrite (FLASHDAT, (unsigned long) dat, FLD_WRLEN);

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);          // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   return retval;                               // return FLASH Pass/Fail
}

//-----------------------------------------------------------------------------------
// FLASH_PageErase
//-----------------------------------------------------------------------------------
// This routine performs an erase of the page in which <addr> is contained.
// This routine assumes that no FLASH operations are currently in progress.
// This routine exits with no FLASH operations currently in progress.
// Returns TRUE if the operation was successful; FALSE otherwise (page protected).
//
int FLASH_PageErase (unsigned int addr)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   bit done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address within page to erase
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x20L, FLCN_LEN);     // set FLASHCON for FLASH Erase
                                                // operation (0x20)

   JTAG_IWrite (FLASHDAT, 0xa5L, FLD_WRLEN);    // set FLASHDAT to 0xa5 to initiate
                                                // erase procedure
```


SILICON LABORATORIES

```
   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);       // set FLASHCON for ‘poll’ operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));        // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);           // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;      // FLFail is next to LSB

   // set return value based on FLFail bit
   return retval;                                // return FLASH Pass/Fail
}
```

Software Examples For the ‘F02x Series

Programming a Single JTAG Device

```
//------------------------------------------------------------------------------------
// JTAG_Flash_F02x.c
//------------------------------------------------------------------------------------
// This program contains some primitive routines which read, write, and erase the FLASH
// through the JTAG port on a C8051Fxxx device under test (DUT).  The JTAG pins on the
// DUT are connected to port pins on the C8051F02x master device.
//
// Target device: C8051F02x
//
// Tool chain: KEIL Eval ‘c’
//

//------------------------------------------------------------------------------------
// Includes
//------------------------------------------------------------------------------------
#include <c8051f020.h>                    // SFR declarations

//------------------------------------------------------------------------------------
// 16-bit SFR Definitions for ‘F02x
//------------------------------------------------------------------------------------

sfr16 DP       = 0x82;                    // data pointer
sfr16 TMR3RL   = 0x92;                    // Timer3 reload value
sfr16 TMR3     = 0x94;                    // Timer3 counter
sfr16 ADC0     = 0xbe;                    // ADC0 data
sfr16 ADC0GT   = 0xc4;                    // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                    // ADC0 less than window
sfr16 RCAP2    = 0xca;                    // Timer2 capture/reload
sfr16 T2       = 0xcc;                    // Timer2
sfr16 RCAP4    = 0xe4;                    // Timer4 capture/reload
sfr16 T4       = 0xf4;                    // Timer4
sfr16 DAC0     = 0xd2;                    // DAC0 data
sfr16 DAC1     = 0xd5;                    // DAC1 data

//------------------------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------------------------
sbit  LED = P1^6;                         // green LED: ‘1’ = ON; ‘0’ = OFF
sbit  SW2 = P3^7;                         // SW2=’0’ means switch pressed

#define SYSCLK       22118400             // SYSCLK frequency in Hz

// GPIO pins connecting to JTAG pins on device to be programmed (DUT)
sbit   TCK = P3^7;                        // JTAG Test Clock
sbit   TMS = P3^6;                        // JTAG Mode Select
sbit   TDI = P3^5;                        // JTAG Data Input
sbit   TDO = P3^4;                        // JTAG Data Output

#define   TRUE 1
#define   FALSE 0

// JTAG Instruction Register Addresses
#define   INST_LENGTH 16                  // number of bits in the Instruction Register
```

```
#define   BYPASS      0xffff
#define   EXTEST      0x0000
#define   SAMPLE      0x0002

#define   RESET       0x2fff              // System RESET Instruction

#define   IDCODE      0x1004              // IDCODE Instruction address/HALT
#define   IDCODE_LEN  32                  // number of bits in the ID code

#define   FLASHCON    0x4082              // FLASH Control Instruction address
#define   FLCN_LEN    8                   // number of bits in FLASHCON

#define   FLASHDAT    0x4083              // FLASH Data Instruction address
#define   FLD_RDLEN   10                  // number of bits in an FLASHDAT read
#define   FLD_WRLEN   8                   // number of bits in an FLASHDAT write

#define   FLASHADR    0x4084              // FLASH Address Instruction address
#define   FLA_LEN     16                  // number of bits in FLASHADR

#define   FLASHSCL    0x4085              // FLASH Scale Instruction address
#define   FLSC_LEN    8                   // number of bits in FLASHSCL

//------------------------------------------------------------------------------------
// Function PROTOTYPES
//------------------------------------------------------------------------------------

void SYSCLK_Init (void);
void PORT_Init (void);

void JTAG_StrobeTCK (void);
void JTAG_Reset (void);
unsigned int JTAG_IR_Scan (unsigned int instruction, int num_bits);
unsigned long JTAG_DR_Scan (unsigned long dat, int num_bits);
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits);
unsigned long JTAG_IRead (unsigned int ireg, int num_bits);
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat);
int FLASH_ByteWrite (unsigned int addr, unsigned char dat);
int FLASH_PageErase (unsigned int addr);


//------------------------------------------------------------------------------------
// MAIN Routine

void main (void) {

   unsigned long id;
   unsigned char dest;
   int pass;

   id = 0x12345678L;

   WDTCN = 0xde;                              // disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                              // initialize crossbar and GPIO
   SYSCLK_Init ();                            // initialize oscillator


   JTAG_Reset ();                             // Reset the JTAG state machine on DUT
```

```
   JTAG_IR_Scan (RESET, INST_LENGTH);          // Reset the DUT

   JTAG_IR_Scan (IDCODE, INST_LENGTH);         // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);       // read the IDCODE
                                               // IDCODE should = 0x10000243 for
                                               // C8051F000 rev D device

   // here we erase the FLASH page 0x1000 - 0x11ff, read 0x1000 (it’s an 0xff),
   // write a 0x66 to 0x1000, and read 0x1000 again (it’s changed to an 0x66).
   while (1) {
      pass = FLASH_PageErase (0x7c00);         // erase page prior to writing...
      while (!pass);                           // handle Write Lock condition

      dest = 0x5a;                             // set test variable to non-0xff value

      pass = FLASH_ByteRead (0x7c00, &dest);   // dest should return 0xff
      while (!pass);                           // handle Read Lock condition

      dest = 0x66;
      pass = FLASH_ByteWrite (0x7c00, dest);   // store 0x66 at 0x1000
      while (!pass);                           // handle Read Lock condition

      pass = FLASH_ByteRead (0x7c00, &dest);   // dest should return 0x66
      while (!pass);                           // handle Read Lock condition

      pass = FLASH_PageErase (0x7c00);
      while (!pass);

      pass = FLASH_ByteRead (0x7c00, &dest);
      while (!pass);
   }
}

//-----------------------------------------------------------------------------
// Functions and Procedures
//-----------------------------------------------------------------------------

//------------------------------------------------------------------------
// SYSCLK_Init
//------------------------------------------------------------------------
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
   int i;                              // delay counter

   OSCXCN = 0x67;                      // start external oscillator with
                                       // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;           // XTLVLD blanking interval (>1ms)

   while (!(OSCXCN & 0x80)) ;          // Wait for crystal osc. to settle

   OSCICN = 0x88;                      // select external oscillator as SYSCLK
                                       // source and enable missing clock
                                       // detector
```

```
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
   XBR0    = 0x04;                     // Enable UART0
   XBR1    = 0x00;
   XBR2    = 0x40;                     // Enable crossbar and weak pull-ups
   P0MDOUT |= 0x01;                    // enable TX0 as a push-pull output
   P1MDOUT |= 0x40;                    // enable P1.6 (LED) as push-pull output

   P3MDOUT |= 0xe0;                    // make P3.7-5 push-pull outputs
   P3 &= ~0xe0;                        // TCK, TMS, and TDI all low
}

//------------------------------------------------------------------------------------
// JTAG_StrobeTCK
//------------------------------------------------------------------------------------
// This routine strobes the TCK pin (brings high then back low again)
// on the target system.
//
void JTAG_StrobeTCK (void) {

   TCK = 1;
   TCK = 0;
}

//------------------------------------------------------------------------------------
// JTAG_Reset
//------------------------------------------------------------------------------------
// This routine places the JTAG state machine on the target system in
// the Test Logic Reset state by strobing TCK 5 times while leaving
// TMS high.  Leaves the JTAG state machine in the Run_Test/Idle state.
//
void JTAG_Reset (void) {

   TMS = 1;

   JTAG_StrobeTCK ();                     // move to Test Logic Reset state
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();

   TMS = 0;

   JTAG_StrobeTCK ();                     // move to Run_Test/Idle state
}

//------------------------------------------------------------------------------------
// JTAG_IR_Scan
//------------------------------------------------------------------------------------
// This routine loads the supplied <instruction> of <num_bits> length into the JTAG
// Instruction Register on the target system.  Leaves in the Run_Test/Idle state.
```

```
// The return value is the n-bit value read from the IR.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
//
unsigned int JTAG_IR_Scan (unsigned int instruction, int num_bits) {

   unsigned int retval;                    // JTAG instruction read
   int   i;                                // JTAG IR bit counter

   retval = 0x0;

   TMS = 1;
   JTAG_StrobeTCK ();                      // move to SelectDR
   TMS = 1;
   JTAG_StrobeTCK ();                      // move to SelectIR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Capture_IR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Shift_IR state

   for (i=0; i < num_bits; i++) {

      TDI = (instruction & 0x01);          // shift IR, LSB-first
      instruction = instruction >> 1;

      retval = retval >> 1;
      if (TDO) {
         retval |= (0x01 << (num_bits - 1));
      }

      if (i == (num_bits - 1)) {
         TMS = 1;                          // move to Exit1_IR state
      }

      JTAG_StrobeTCK();
   }

   TMS = 1;
   JTAG_StrobeTCK ();                      // move to Update_IR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to RTI state

   return retval;
}

//-----------------------------------------------------------------------------------
// JTAG_DR_Scan
//-----------------------------------------------------------------------------------
// This routine shifts <num_bits> of <data> into the Data Register, and returns
// up to 32-bits of data read from the Data Register.
// Leaves in the Run_Test/Idle state.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
//
unsigned long JTAG_DR_Scan (unsigned long dat, int num_bits) {

   unsigned long   retval;                 // JTAG return value
   int   i;                                // JTAG DR bit counter

   retval = 0x0L;
```

```
   TMS = 1;
   JTAG_StrobeTCK ();                      // move to SelectDR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Capture_DR
   TMS = 0;
   JTAG_StrobeTCK ();                      // move to Shift_DR state

   for (i=0; i < num_bits; i++) {

      TDI = (dat & 0x01);                  // shift DR, LSB-first
      dat = dat >> 1;

      retval = retval >> 1;
      if (TDO) {
         retval |= (0x01L << (num_bits - 1));
      }

      if ( i == (num_bits - 1)) {
         TMS = 1;                           // move to Exit1_DR state
      }

      JTAG_StrobeTCK();
   }
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to Update_DR
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to RTI state

   return retval;
}

//------------------------------------------------------------------------------------
// JTAG_IWrite
//------------------------------------------------------------------------------------
// This routine performs an indirect write to register <ireg>, containing <dat>, of
// <num_bits> in length.  It follows the write operation with a polling operation, and
// returns when the operation is completed.  Note: the polling implemented here refers
// to the JTAG register write operation being completed, NOT the FLASH write operation.
// Polling for the FLASH write operation is handled at a higher level
// Examples of valid indirect registers are:
//  FLCN - FLASH Control
//  FLSC - FLASH Scale
//  FLA - FLASH Address
//  FLD - FLASH Data
// Leaves in the Run_Test/Idle state.
//
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits) {

   int done;                               // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);       // load IR with <ireg>

   dat |= (0x03L << num_bits);             // append 'WRITE' opcode to data

   // load DR with <dat>
   JTAG_DR_Scan (dat, num_bits + 2);       // initiate the JTAG write

   // load DR with '0', and check for BUSY bit to go to '0'.
   do {
```

```
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);
}

//-----------------------------------------------------------------------------------
// JTAG_IRead
//-----------------------------------------------------------------------------------
// This routine performs an indirect read of register <ireg>, of <num_bits> in length.
// It follows the read operation with a polling operation, and returns when the
// operation is completed.  Note: the polling implemented here refers to the JTAG
// register read operation being completed, NOT the FLASH read operation.
// Polling for the FLASH read operation is handled at a higher level.
// Examples of valid indirect registers are:
//  FLCN - FLASH Control
//  FLSC - FLASH Scale
//  FLA - FLASH Address
//  FLD - FLASH Data
// Leaves in the Run_Test/Idle state.
//
unsigned long JTAG_IRead (unsigned int ireg, int num_bits) {

   unsigned long retval;                 // value returned from READ operation
   int done;                             // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);     // load IR with <ireg>

   // load DR with read opcode (0x02)

   JTAG_DR_Scan (0x02L, 2);              // initiate the JTAG read

   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);

   retval = JTAG_DR_Scan (0x0L, num_bits + 1);  // allow poll operation to
                                                //  read remainder of the bits
   retval = retval >> 1;                 // shift JTAG_BUSY bit off the end

   return retval;
}

//-----------------------------------------------------------------------------------
// FLASH_ByteRead
//-----------------------------------------------------------------------------------
// This routine reads the byte at <addr> and stores it at the address pointed to by
// <pdat>.
// Returns TRUE if the operation was successful; FALSE otherwise (page read-protected).
//
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   int done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                //  frequency (2MHz = 0x86)

   // set FLASHADR to address to read from
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);
```

```
   JTAG_IWrite (FLASHCON, 0x02L, FLCN_LEN);     // set FLASHCON for FLASH Read
                                                //  operation (0x02)

   JTAG_IRead (FLASHDAT, FLD_RDLEN);            // initiate the read operation

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBUSY to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, FLD_RDLEN);  // read the resulting data

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   testval = testval >> 2;                      // shift data.0 into LSB position

   *pdat = (unsigned char) testval;             // place data in return location

   return retval;                               // return FLASH Pass/Fail
}

//------------------------------------------------------------------------------------
// FLASH_ByteWrite
//------------------------------------------------------------------------------------
// This routine writes the data <dat> to FLASH at the address <addr>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
// write-protected).
//
int FLASH_ByteWrite (unsigned int addr, unsigned char dat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   int done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address to write to
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x10L, FLCN_LEN);     // set FLASHCON for FLASH Write
                                                //  operation (0x10)

   // initiate the write operation
   JTAG_IWrite (FLASHDAT, (unsigned long) dat, FLD_WRLEN);

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);          // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   return retval;                               // return FLASH Pass/Fail
```

```
}

//-----------------------------------------------------------------------------------
// FLASH_PageErase
//-----------------------------------------------------------------------------------
// This routine performs an erase of the page in which <addr> is contained.
// This routine assumes that no FLASH operations are currently in progress.
// This routine exits with no FLASH operations currently in progress.
// Returns TRUE if the operation was successful; FALSE otherwise (page protected).
//
int FLASH_PageErase (unsigned int addr)
{
   unsigned long testval;                        // holds result of FLASHDAT read
   int done;                                     // TRUE/FALSE flag
   int retval;                                   // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);      // set FLASHSCL based on SYSCLK
                                                 //  frequency (2MHz = 0x86)

   // set FLASHADR to address within page to erase
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x20L, FLCN_LEN);      // set FLASHCON for FLASH Erase
                                                 //  operation (0x20)

   JTAG_IWrite (FLASHDAT, 0xa5L, FLD_WRLEN);     // set FLASHDAT to 0xa5 to initiate
                                                 //  erase procedure

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);       // set FLASHCON for ‘poll’ operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));        // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);           // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;      // FLFail is next to LSB

   // set return value based on FLFail bit
   return retval;                                // return FLASH Pass/Fail
}
```

Programming Multiple JTAG Devices in a Chain

```
//************************************************************************************
// JTAG_Chain_F02x.c
//------------------------------------------------------------------------------------
// This program contains some primitive routines which gather information through the
// JTAG port on multiple JTAG compatible devices under test (DUT) connected in a
// chain. The TCK & TMS JTAG pins on the DUT are connected in parallel to port pins on
// the C8051F02x master device and the TDI & TDO pins are connected in
// series.
//
// **NOTE: The first device in the chain (device 0) is the one whose TDO pin is
//         connected to the TDO pin of the master device.
//
// Target device: C8051F02x
//
// Tool chain: KEIL Eval 'c'
//************************************************************************************

//------------------------------------------------------------------------------------
// Includes
//------------------------------------------------------------------------------------

#include <c8051f020.h>                    // SFR declarations

//------------------------------------------------------------------------------------
// 16-bit SFR Definitions for 'F02x
//------------------------------------------------------------------------------------

sfr16 DP       = 0x82;                    // data pointer
sfr16 TMR3RL   = 0x92;                    // Timer3 reload value
sfr16 TMR3     = 0x94;                    // Timer3 counter
sfr16 ADC0     = 0xbe;                    // ADC0 data
sfr16 ADC0GT   = 0xc4;                    // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                    // ADC0 less than window
sfr16 RCAP2    = 0xca;                    // Timer2 capture/reload
sfr16 T2       = 0xcc;                    // Timer2
sfr16 RCAP4    = 0xe4;                    // Timer4 capture/reload
sfr16 T4       = 0xf4;                    // Timer4
sfr16 DAC0     = 0xd2;                    // DAC0 data
sfr16 DAC1     = 0xd5;                    // DAC1 data

//------------------------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------------------------

#define MAX_NUM_DEVICES_IN_CHAIN 10

sbit  LED = P1^6;                         // green LED: '1' = ON; '0' = OFF
sbit  SW2 = P3^7;                         // SW2='0' means switch pressed

#define SYSCLK       22118400             // SYSCLK frequency in Hz

sbit   TCK = P3^7;      // JTAG Test Clock -- Connected to TCK pin on all devices.
sbit   TMS = P3^6;      // JTAG Mode Select -- Connected to TMS pin on all devices.
sbit   TDI = P3^5;      // JTAG Data Input(output of master) -- Connected to the
                        //      TDI pin of device n.
sbit   TDO = P3^4;      // JTAG Data Output (input to master)-- Connected to the
                        //      TDO pin of device 0.
```

```
#define   TRUE 1
#define   FALSE 0

// JTAG Instruction Register Addresses
#define   INST_LENGTH 16                   // number of bits in the C8051Fxxx
#define   BYPASS      0xffff               // Instruction Register
#define   EXTEST      0x0000
#define   SAMPLE      0x0002

#define   RESET       0x2fff               // System RESET Instruction

#define   IDCODE      0x1004               // IDCODE Instruction address/HALT
#define   IDCODE_LEN  32                   // number of bits in the ID code

#define   FLASHCON    0x4082               // FLASH Control Instruction address
#define   FLCN_LEN    8                    // number of bits in FLASHCON

#define   FLASHDAT    0x4083               // FLASH Data Instruction address
#define   FLD_RDLEN   10                   // number of bits in an FLASHDAT read
#define   FLD_WRLEN   8                    // number of bits in an FLASHDAT write

#define   FLASHADR    0x4084               // FLASH Address Instruction address
#define   FLA_LEN     16                   // number of bits in FLASHADR

#define   FLASHSCL    0x4085               // FLASH Scale Instruction address
#define   FLSC_LEN    8                    // number of bits in FLASHSCL

//-----------------------------------------------------------------------------------
// Global Variable DECLARATIONS
//-----------------------------------------------------------------------------------

// The addresses of the following variables are explicitly defined for viewing
// purposes. If the width of the external memory window is 5 bytes, then each
// device will take up exactly one row starting from the second row.
char xdata num_devices  _at_ 0x0000;

char xdata num_devices_before _at_ 0x0001; // #devices before and after the isolated
char xdata num_devices_after  _at_ 0x0002; // device
char xdata num_IR_bits_before _at_ 0x0003; // #instruction register bits before and
char xdata num_IR_bits_after  _at_ 0x0004; // after the isolated device


typedef struct JTAG_Information {          // Discovery information
      unsigned char IR_length;             // Instruction register length
      unsigned long id;                    // Identification code for each device
} JTAG_Information;
                                           // Array: one entry per device in the
                                           // JTAG chain
JTAG_Information xdata JTAG_info[MAX_NUM_DEVICES_IN_CHAIN];

//-----------------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------------

void SYSCLK_Init (void);
void PORT_Init (void);

void JTAG_StrobeTCK (void);
```

```
void JTAG_Reset (void);

void Blink_Led(void);

void init(void);
void JTAG_Discover(void);
void JTAG_Discover_IR(void);
void JTAG_Discover_DR(void);
void JTAG_Isolate(char index);

unsigned long JTAG_IR_Scan (unsigned long instruction, char num_bits) ;
unsigned long JTAG_DR_Scan (unsigned long dat, char num_bits);

void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits);
unsigned long JTAG_IRead (unsigned int ireg, int num_bits);
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat);
int FLASH_ByteWrite (unsigned int addr, unsigned char dat);
int FLASH_PageErase (unsigned int addr);

//------------------------------------------------------------------------------------
// MAIN Routine
//------------------------------------------------------------------------------------
void main (void)
{

   long xdata id;
   unsigned char dest;
   int pass;
   int address;
   char device = 0;


   WDTCN = 0xde;                           // disable watchdog timer
   WDTCN = 0xad;

   PORT_Init ();                           // initialize crossbar and GPIO
   SYSCLK_Init ();                         // initialize oscillator

   LED = 1;                                // turn on the LED

   init();                                 // initialize JTAG Chain variables
   JTAG_Discover();                        // IDCODE should = 0x10000243 for
                                           // C8051F000 rev D device

   JTAG_Isolate(0);                        // isolate device 0
   JTAG_IR_Scan (IDCODE, INST_LENGTH);     // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);   // get the ID Code of the isolated device

   // comment out this code if you have less than two devices in the chain
   JTAG_Isolate(1);
   JTAG_IR_Scan (IDCODE, INST_LENGTH);     // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);   // get the ID Code of the isolated device

   // comment out this code if you have less than three devices in the chain
   JTAG_Isolate(2);
   JTAG_IR_Scan (IDCODE, INST_LENGTH);     // load IDCODE into IR and HALT the DUT
   id = JTAG_DR_Scan (0x0L, IDCODE_LEN);   // get the ID Code of the isolated device
```

```
   for(device = 0; device < num_devices; device++) {

      JTAG_Isolate(device);

      //TEST 1 -- ERASE A FLASH PAGE
      pass = FLASH_PageErase (0x1000);       // erase page prior to writing
      while (!pass);                         // handle Write Lock condition


      //Verify that locations 0x1000 - 0x11FF are 0xFF
      for(address = 0x1000; address < 0x1200; address++){
         pass = FLASH_ByteRead (address, &dest);   // dest should return 0xff
         if(!pass || dest != 0xFF) Blink_Led();
      }

      //TEST 2 -- WRITE A PATTERN TO FLASH PAGE

      for(address = 0x1000; address < 0x1200; address++){
         dest = address & 0x00FF;               // strip away upper 8 bits
         pass = FLASH_ByteWrite (address, dest);// store LSByte of address at address
         while (!pass);                         // handle Read Lock condition
      }

      dest = 0x12;                              // set test variable to non-0xff value

      //Verify that locations 0x1000 - 0x11FF are following the pattern
      for(address = 0x1000; address < 0x1200; address++){
         pass = FLASH_ByteRead (address, &dest);
         if(!pass || dest != (address & 0x00FF)) Blink_Led();
      }
   }


   LED = 0;                                     // turn off the led,
                                                // program executed correctly
   while(1);
}

//************************************************************************************
// Function and Procedure DEFINITIONS
//************************************************************************************
//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
   int i;                              // delay counter

   OSCXCN = 0x67;                      // start external oscillator with
                                       // 22.1184MHz crystal

   for (i=0; i < 256; i++) ;           // XTLVLD blanking interval (>1ms)

   while (!(OSCXCN & 0x80)) ;          // Wait for crystal osc. to settle
```

```
   OSCICN = 0x88;                       // select external oscillator as SYSCLK
                                        // source and enable missing clock
                                        // detector
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
   XBR0    = 0x04;                      // Enable UART0
   XBR1    = 0x00;
   XBR2    = 0x40;                      // Enable crossbar and weak pull-ups
   P0MDOUT |= 0x01;                     // enable TX0 as a push-pull output
   P1MDOUT |= 0x40;                     // enable P1.6 (LED) as push-pull output

   P3MDOUT |= 0xe0;                     // make P3.7-5 push-pull outputs
   P3 &= ~0xe0;                         // TCK, TMS, and TDI all low
}
//------------------------------------------------------------------------------------
// Blink_Led
//------------------------------------------------------------------------------------
// This routine blinks the Green LED forever to indicate an error.
//
void Blink_Led(void)
{
   int i;                                     // millisecond counter
   int ms = 200;                              // stay in each state for ms milliseconds

   TCON  &= ~0x30;                            // STOP Timer0 and clear overflow flag
   TMOD  &= ~0x0F;                            // configure Timer0 to 16-bit mode
   TMOD  |=  0x01;
   CKCON |=  0x08;                            // Timer0 counts SYSCLKs


   while (1){

   LED = ~LED;

      for (i = 0; i < ms; i++) {              // count milliseconds
         TR0 = 0;                             // STOP Timer0
         TH0 = (-SYSCLK/1000) >> 8;           // SET Timer0 to overflow in 1ms
         TL0 = -SYSCLK/1000;
         TR0 = 1;                             // START Timer0

         while(TF0 == 0);                     // wait for overflow

         TF0 = 0;                             // clear overflow indicator
      }
   }
}

//------------------------------------------------------------------------------------
// init
//------------------------------------------------------------------------------------
```

```
// This routine initializes the variables used in a JTAG chain.
//
void init (void)
{

   num_devices = 1;                           // The default number of devices is one.
                                              // JTAG_Discover() does not have to be
                                              // called if only one device is connected.

   num_devices_before = 0;                    // Initializing these variables to zero
   num_devices_after = 0;                     // allows calling the JTAG_IR_Scan() and
   num_IR_bits_before = 0;                    // the JTAG_DR_Scan() without first
   num_IR_bits_after = 0;                     // calling JTAG_Isolate() when there is
                                              // only one device in the chain.
}

//-----------------------------------------------------------------------------------
// JTAG_StrobeTCK
//-----------------------------------------------------------------------------------
// This routine strobes the TCK pin (brings high then back low again)
// on the target system.
//
void JTAG_StrobeTCK (void)
{

   TCK = 1;
   TCK = 0;
}

//-----------------------------------------------------------------------------------
// JTAG_Reset
//-----------------------------------------------------------------------------------
// This routine places the JTAG state machine on the target system in
// the Test Logic Reset state by strobing TCK 5 times while leaving
// TMS high.  Leaves the JTAG state machine in the Run_Test/Idle state.
//
void JTAG_Reset (void)
{

   TMS = 1;

   JTAG_StrobeTCK ();                         // move to Test Logic Reset state
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();
   JTAG_StrobeTCK ();

   TMS = 0;

   JTAG_StrobeTCK ();                         // move to Run_Test/Idle state
}

//-----------------------------------------------------------------------------------
// JTAG_Discover
//-----------------------------------------------------------------------------------
// This routine sequentially queries a chain of JTAG devices and accomplishes the
// following three tasks.
//    For the global struct array <JTAG_info>
//      -- fills in the length of each device's instruction register
```

```
//      -- fills in each device's IDCODE.
//    For the global variable <num_devices>
//      -- updates it with the number of JTAG devices connected in the chain.
//
void JTAG_Discover(void)
{

   JTAG_Discover_IR();

   // At this point we know num_devices(a global variable) and we know the
   // length of each device's IR given in the variable JTAG_info[].IR_length

   JTAG_Discover_DR();                       // Read and assign the ID for each
                                             // device

} //end discover

//-------------------------------------------------------------------------------------
// JTAG_Discover_IR
//-------------------------------------------------------------------------------------
// This routine fills a structure with the length of each device's instruction
// register. It also updates the global variable <num_devices> with the number of
// JTAG devices connected in the chain.
//
// BACKGROUND: When an IRSCAN is issued, a JTAG device must return a 1 as the LSB
//             and zeros in all the other bits.  We shift in all ones so when we
//             encounter two ones in a row, we know we are past the end of the chain.
//             A state machine is implemented in this routine to keep track of
//             inputs received.
//
// STATE DEFINITONS:
//             0 - NO INPUTS -- at beginning of chain
//          1 - INPUT SEQUENCE: 1 -- could be at a new device or at chain end
//             2 - INPUT SEQUENCE: 100..0 -- counting zeros
//
//
void JTAG_Discover_IR(void)
{

   char state = 0;                          // beginning of chain

   char num_zeros = 0;                      // number of zeros following a one in
                                            // an IR_SCAN. num_zeros + 1 = IR_length

   char current_device_index = -1;          // current_device_index + 1 = num_devices
                                            // (on the last iteration)
   bit done = FALSE;                        // TRUE when end of chain is reached

   JTAG_Reset();                            // RESET and move to Run_Test/Idle

   // advance to Shift_IR State
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to SelectDR
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to SelectIR
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to Capture_IR
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to Shift_IR state and get the
```

```
                                          // the first input

   TDI = 1;                               // STATE is initially 0
                                          // shift in all ones
   // for each device
   do{

      if(TDO != 1){                       // Error if the first input is not one.
         Blink_Led();                     // Could mean bad connections or
      }                                   // non-compliant devices.



      state = 1;                          // received a 1, could be at a new
                                          // device or at the end of the chain

      num_zeros = 0;                      // initialize for the zero counting loop

      // for the number of zeros in each device's IR
      do {

         JTAG_StrobeTCK();                // get the next bit.

         switch(state){

            case 1: if(TDO == 0){         // found new device(10)
                       current_device_index++;
                       num_zeros++;
                       state = 2;
                    } else {
                       done = TRUE;       // at end of chain (11)
                    }
                    break;

            case 2: if(TDO == 0){
                      num_zeros++;        // counting zeros (10..0)
                    } else {
                      state = 1;          // past end of current device (10..01)
                    }
                    break;

            default: Blink_Led();         // an error has occurred

         } // end switch


      } while ((state != 1) && (!done));  // while the input is not one,
                                          // count zeros until we get a one.

      if (!done) {                        // if we are not past the last device

         JTAG_info[current_device_index].IR_length = num_zeros + 1;
      }

   } while (!done);                       //while we are not past the last device

   num_devices = current_device_index + 1;

   // navigate the JTAG State Machine back to RTI state.
```

```
   TMS = 1;
   JTAG_StrobeTCK ();                         // move to Exit1_IR state
   TMS = 1;
   JTAG_StrobeTCK ();                         // move to Update_IR state
   TMS = 0;
   JTAG_StrobeTCK ();                         // move to Run_Test/Idle state

}

//-------------------------------------------------------------------------------------
// JTAG_Discover_DR
//-------------------------------------------------------------------------------------
//GOAL: Obtain the ID code of each device(If it supports IDCODE), and fill in
//      the field JTAG_info[].id (32-bit).
//      Assign all zeros if device does not have an IDCODE.
//
//BACKGROUND: After JTAG State Machine Reset, the IDCODE is automatically selected
//            If a device does not have an IDCODE register, the BYPASS
//            register is selected instead.
//            On a DR_SCAN, each IDCODE register returns a 32-bit ID with LSB = 1
//            and each BYPASS register returns 1-bit = 0.

void JTAG_Discover_DR(void)
{

   char current_device_index = 0;

   unsigned char i;                         // loop counter


   JTAG_Reset ();                           // Reset the JTAG state machine on DUT
                                            // move to Run_Test/Idle

   // The IDCODE or the BYPASS Register is automatically selected.

   // Navigate to the Shift_DR state
   TMS = 1;
   JTAG_StrobeTCK ();                       // move to SelectDR
   TMS = 0;
   JTAG_StrobeTCK ();                       // move to Capture_DR

   TMS = 0;
   TDI = 1;                                 // shift in all ones

   current_device_index = 0;

   while (current_device_index < num_devices) {

      JTAG_StrobeTCK ();                    // move to Shift_DR state and get input

      if (TDO == 0) {                       // Device does not have an IDCODE register

         JTAG_info[current_device_index].id = 0x00000000L;

      } else { // TDO == 1

         JTAG_info[current_device_index].id = 0x80000000L;

         for (i = 0; i < 31; i++){          // Get the next 31-bits of the device ID
```

```
            JTAG_StrobeTCK ();

            JTAG_info[current_device_index].id =
              JTAG_info[current_device_index].id >> 1;

            if (TDO) {
               JTAG_info[current_device_index].id |= 0x80000000L;
            }
         } // end for

      } // end if-else

      current_device_index++;

   } // end while

   //fill the rest of the entries with zeros
   for (; current_device_index < MAX_NUM_DEVICES_IN_CHAIN; current_device_index++) {

      JTAG_info[current_device_index].IR_length = 0;
      JTAG_info[current_device_index].id = 0x00000000L;
   }

   // Navigate JTAG State Machine back to RTI state
   TMS = 1;
   JTAG_StrobeTCK ();                     // move to Exit1_DR
   TMS = 1;
   JTAG_StrobeTCK ();                     // move to Update DR
   TMS = 0;
   JTAG_StrobeTCK ();                     // move to RTI

}

//------------------------------------------------------------------------------------
// JTAG_Isolate
//------------------------------------------------------------------------------------
// This routine updates 4 global variables.  JTAG_Discover() must be called prior to
// calling this routine in order to set up the data structure.
//
// VARIABLE DEFINITIONS
//    num_IR_bits_before -- number of instruction register bits before the isolated
//                          device
//    num_IR_bits_after  -- number of instruction register bits after the isolated
//                          device
//    num_devices_before -- number of devices before the isolated device
//    num_devices_after  -- number of device after the isolated device
//
void JTAG_Isolate(char index)
{

   unsigned char i;

   if ((index > (num_devices - 1)) || (index < 0) ) {
                                          // check if index is out of range
      Blink_Led();
   }

   num_devices_before = index;
```

```
   num_devices_after = num_devices - index - 1;

   num_IR_bits_before = 0;                // initializing for loop
   num_IR_bits_after = 0;

   for (i = 0; i < num_devices; i++) {

      if (i < index) {

         num_IR_bits_before += JTAG_info[i].IR_length;

      } else if (i > index) {

         num_IR_bits_after += JTAG_info[i].IR_length;
      }

      // last case -- equal, do nothing
   } // end for
} //end isolate


//-----------------------------------------------------------------------------------
// JTAG_IR_Scan
//-----------------------------------------------------------------------------------
// This routine loads the supplied <instruction> of <num_bits> length into the JTAG
// Instruction Register on the isolated device. It shifts the BYPASS opcode (all ones)
// into the Instruction Registers of the other devices in the chain.
//
// NOTE: JTAG_Discover() must be called before this function is called.
//
// NOTE: If more than one device is connected in the chain, JTAG_Isolate() must also
//       be called prior to calling this function.
//
// The return value is the n-bit value read from the IR.
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
// Leaves JTAG in the Run_Test/Idle state.
//
unsigned long JTAG_IR_Scan (unsigned long instruction, char num_bits)
{

    unsigned long retval;                        // JTAG instruction read
    char i;                                      // JTAG IR bit counter

    retval = 0x0L;

    // navigate the JTAG State Machine in all devices to the Shift_IR state
    TMS = 1;
    JTAG_StrobeTCK ();                           // move to SelectDR
    TMS = 1;
    JTAG_StrobeTCK ();                           // move to SelectIR
    TMS = 0;
    JTAG_StrobeTCK ();                           // move to Capture_IR
    TMS = 0;
    JTAG_StrobeTCK ();                           // move to Shift_IR state

    TDI=1;
    for (i=0; i < num_IR_bits_before; i++) {
```

```
        JTAG_StrobeTCK();                           // fill the IR of the devices
                                                    // before the isolated device
    }                                               // with all ones, the BYPASS opcode


    for (i=0; i < num_bits; i++) {

        TDI = (instruction & 0x01);                 // determine output
        instruction = instruction >> 1;

        retval = retval >> 1;
        if (TDO) {
            retval |= (0x01 << (num_bits - 1));
        }

        if ((i == (num_bits - 1)) && (num_IR_bits_after == 0)) {
            TMS = 1;                                // move to Exit1_IR state
        }

        JTAG_StrobeTCK();                           // move to Shift_IR state
                                                    // advance
    }

    TDI = 1;
    for (i=0; i < num_IR_bits_after; i++) {         // now process IR bits after the
                                                    // isolated device
        if (i == (num_IR_bits_after - 1)) {
            TMS = 1;                                // move to Exit1_IR state
        }


        JTAG_StrobeTCK();                           // fill the IR of the devices
                                                    // after the isolated device
    }                                               // with all ones, the BYPASS opcode.

    // navigate back to the RTI state

    TMS = 1;
    JTAG_StrobeTCK ();                              // move to Update_IR
    TMS = 0;
    JTAG_StrobeTCK ();                              // move to RTI state

    return retval;
}

//------------------------------------------------------------------------------------
// JTAG_DR_Scan
//------------------------------------------------------------------------------------
// This routine shifts <num_bits> of <data> into the Data Register of the isolated
// device in the chain, and returns up to 32-bits of data read from its Data Register.
//
// Assumes the JTAG state machine starts in the Run_Test/Idle state.
// Leaves in the Run_Test/Idle state.
//
unsigned long JTAG_DR_Scan (unsigned long dat, char num_bits)
{

    unsigned long retval;                          // JTAG return value
    char i;                                        // JTAG DR bit counter
```

```
    retval = 0x0L;

    // navigate the JTAG State Machine in all devices to the Shift_DR state
    TMS = 1;
    JTAG_StrobeTCK ();                             // move to SelectDR
    TMS = 0;
    JTAG_StrobeTCK ();                             // move to Capture_DR
    TMS = 0;
    JTAG_StrobeTCK ();                             // move to Shift_DR state


    TDI = 0;
    for (i=0; i < num_devices_before; i++) {

       JTAG_StrobeTCK();                           // fill the BYPASS Register
                                                   // of the devices before the
    }                                              // isolated device with zeros.



    for (i=0; i < num_bits; i++) {

        TDI = (dat & 0x01);                        // determine the output
        dat = dat >> 1;

        retval = retval >> 1;
        if (TDO) {
            retval |= (0x01L << (num_bits - 1));
        }

        if ((i == (num_bits - 1)) && (num_devices_after == 0)) {
            TMS = 1;                               // move to Exit1_IR state
        }

        JTAG_StrobeTCK();
                                                   //output and get input
    }

    TDI = 0;
    for (i=0; i < num_devices_after; i++) {

        if (i == (num_devices_after - 1)) {
            TMS = 1;                               // move to Exit1_IR state
        }

                                                   // move to Shift_DR state,
       JTAG_StrobeTCK();                           // fill the BYPASS Register
                                                   // of the devices after the
    }                                              // isolated device with zeros.

    // navigate the JTAG State Machine in all devices to the RTI state
    TMS = 1;
    JTAG_StrobeTCK ();                             // move to Update_DR
    TMS = 0;
    JTAG_StrobeTCK ();                             // move to RTI state

    return retval;                                 // retval is MSB aligned
}
```

```
//-----------------------------------------------------------------------------------
// JTAG_IWrite
//-----------------------------------------------------------------------------------
// This routine performs an indirect write to register <ireg>, containing <dat>, of
// <num_bits> in length.  It follows the write operation with a polling operation, and
// returns when the operation is completed.  Note: the polling implemented here refers
// to the JTAG register write operation being completed, NOT the FLASH write operation.
// Polling for the FLASH write operation is handled at a higher level
// Examples of valid indirect registers are:
//  FLASHCON - FLASH Control
//  FLASHSCL - FLASH Scale
//  FLASHADR - FLASH Address
//  FLASHDAT - FLASH Data
// Leaves in the Run_Test/Idle state.
//
void JTAG_IWrite (unsigned int ireg, unsigned long dat, int num_bits)
{

   bit done;                             // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);      // load IR with <ireg>

   dat |= (0x03L << num_bits);            // append 'WRITE' opcode to data

   // load DR with <dat>
   JTAG_DR_Scan (dat, num_bits + 2);      // initiate the JTAG write

   // load DR with '0', and check for BUSY bit to go to '0'.
   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);
}

//-----------------------------------------------------------------------------------
// JTAG_IRead
//-----------------------------------------------------------------------------------
// This routine performs an indirect read of register <ireg>, of <num_bits> in length.
// It follows the read operation with a polling operation, and returns when the
// operation is completed.  Note: the polling implemented here refers to the JTAG
// register read operation being completed, NOT the FLASH read operation.
// Polling for the FLASH read operation is handled at a higher level.
// Examples of valid indirect registers are:
//  FLASHCON - FLASH Control
//  FLASHSCL - FLASH Scale
//  FLASHADR - FLASH Address
//  FLASHDAT - FLASH Data
// Leaves JTAG in the Run_Test/Idle state.
//
unsigned long JTAG_IRead (unsigned int ireg, int num_bits) {

   unsigned long retval;                 // value returned from READ operation
   bit done;                             // TRUE = write complete; FALSE otherwise

   JTAG_IR_Scan (ireg, INST_LENGTH);      // load IR with <ireg>

   // load DR with read opcode (0x02)
   JTAG_DR_Scan (0x02L, 2);               // initiate the JTAG read
```

```
   do {
      done = !(JTAG_DR_Scan (0x0L, 1));   // poll for JTAG_BUSY bit
   } while (!done);

   retval = JTAG_DR_Scan (0x0L, num_bits + 1);  // allow poll operation to
                                                // read remainder of the bits
   retval = retval >> 1;                  // shift JTAG_BUSY bit off the end

   return retval;
}

//------------------------------------------------------------------------------------
// FLASH_ByteRead
//------------------------------------------------------------------------------------
// This routine reads the byte at <addr> and stores it at the address pointed to by
// <pdat>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
// read-protected).
//
int FLASH_ByteRead (unsigned int addr, unsigned char *pdat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   bit done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address to read from
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x02L, FLCN_LEN);     // set FLASHCON for FLASH Read
                                                // operation (0x02)

   JTAG_IRead (FLASHDAT, FLD_RDLEN);            // initiate the read operation

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBUSY to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, FLD_RDLEN);  // read the resulting data

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   testval = testval >> 2;                      // shift data.0 into LSB position

   *pdat = (unsigned char) testval;             // place data in return location

   return retval;                               // return FLASH Pass/Fail
}

//------------------------------------------------------------------------------------
// FLASH_ByteWrite
//------------------------------------------------------------------------------------
// This routine writes the data <dat> to FLASH at the address <addr>.
// Returns TRUE if the operation was successful; FALSE otherwise (page
// write-protected).
```

```
//
int FLASH_ByteWrite (unsigned int addr, unsigned char dat)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   int done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address to write to
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x10L, FLCN_LEN);     // set FLASHCON for FLASH Write
                                                // operation (0x10)

   // initiate the write operation
   JTAG_IWrite (FLASHDAT, (unsigned long) dat, FLD_WRLEN);

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation

   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);          // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   return retval;                               // return FLASH Pass/Fail
}

//-----------------------------------------------------------------------------------
// FLASH_PageErase
//-----------------------------------------------------------------------------------
// This routine performs an erase of the page in which <addr> is contained.
// This routine assumes that no FLASH operations are currently in progress.
// This routine exits with no FLASH operations currently in progress.
// Returns TRUE if the operation was successful; FALSE otherwise (page protected).
//
int FLASH_PageErase (unsigned int addr)
{
   unsigned long testval;                       // holds result of FLASHDAT read
   bit done;                                    // TRUE/FALSE flag
   int retval;                                  // TRUE if operation successful

   JTAG_IWrite (FLASHSCL, 0x86L, FLSC_LEN);     // set FLASHSCL based on SYSCLK
                                                // frequency (2MHz = 0x86)

   // set FLASHADR to address within page to erase
   JTAG_IWrite (FLASHADR, (unsigned long) addr, FLA_LEN);

   JTAG_IWrite (FLASHCON, 0x20L, FLCN_LEN);     // set FLASHCON for FLASH Erase
                                                // operation (0x20)

   JTAG_IWrite (FLASHDAT, 0xa5L, FLD_WRLEN);    // set FLASHDAT to 0xa5 to initiate
                                                // erase procedure

   JTAG_IWrite (FLASHCON, 0x0L, FLCN_LEN);      // set FLASHCON for 'poll' operation
```

```
   do {
      done = !(JTAG_IRead (FLASHDAT, 1));       // poll for FLBusy to de-assert
   } while (!done);

   testval = JTAG_IRead (FLASHDAT, 2);          // read FLBusy and FLFail

   retval = (testval & 0x02) ? FALSE: TRUE;     // FLFail is next to LSB

   // set return value based on FLFail bit
   return retval;                               // return FLASH Pass/Fail
}
```

Contact Information

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Email: productinfo@silabs.com
Internet: www.silabs.com

SILICON LABORATORIES