

APPLICATION NOTE 4012

Implementing a JTAG Bootloader Master for the MAXQ2000 Microcontroller

Abstract: The JTAG bootloader provided by MAXQ® microcontrollers allows an external JTAG master to easily identify and program any MAXQ microcontroller using a set of standardized commands. The code included with this application note can be used as a starting point to build a full-featured JTAG bootloader master application. The master application can identify, initialize, load, and verify the code and data memory contents of any MAXQ microcontroller which supports the standardized bootloader command set.

Overview

MAXQ microcontrollers that include rewriteable onboard program memory typically include a ROM-based bootloader that enables the program memory to be loaded using the microcontroller's JTAG-compatible debug port. Although the exact functionality provided by the JTAG bootloader varies from device to device, it typically includes commands which allow program and data memory to be read, written, verified, and erased. Some devices provide alternative interfaces to the bootloader (such as a serial port or SPI™ interface), but the JTAG interface is most commonly used for two reasons. First, the JTAG interface must already be present to support in-circuit debugging functions. Second, the JTAG interface is not generally utilized by the end-user application (unlike a serial port). An optional password mechanism can be used to restrict access to the bootloader or in-circuit debugging features once the program code has been loaded. Consult the product specific information, including data sheets and User's Guides, for detailed information on the functionality supported by each MAXQ device.

This application note covers the basic steps needed to implement a JTAG bootloader master for the [MAXQ2000](#). These steps include interfacing to the JTAG port, communicating with the Test Access Port (TAP) controller, activating bootloader mode, and sending commands to the ROM-based bootloader. Since the JTAG port generally operates the same on all MAXQ devices and since MAXQ bootloaders operate using a shared command set, most of the topics covered in this application note (as well as most of the example code) will apply when implementing a JTAG bootloader master for any MAXQ microcontroller.

Other than a serial port, no special features of the MAXQ2000 were used for this implementation. This means that the example code presented here could easily be retargeted to run on any MAXQ20 device with sufficient program memory. The code was written in MAXQ assembly and compiled using the MAX-IDE development environment. The code is available for [download](#).

Hardware Setup

The example code for this application note was developed using a pair of MAXQ2000 evaluation (EV) kits. Two MAXQ2000 EV kits are needed to execute the software described in here. One MAXQ2000 (the JTAG master) runs the example code; the second MAXQ2000 acts as the JTAG slave which is reprogrammed by the master. Standard 8.00MHz crystals were used on both MAXQ2000 microcontrollers.

The master MAXQ2000 EV kit was modified by installing a 2 x 5 pin header in the prototyping area, which provided the master side of the JTAG cable connection. The pins on the header followed the standard MAXQ JTAG header layout and were connected as shown in **Table 1**.

Table 1. MAXQ2000 JTAG Connections

JTAG Header Pin	JTAG Cable Function	MAXQ2000 JTAG Master Connection	MAXQ2000 JTAG Slave Connection
1	TCK (Test Clock)	P0.0 (Output)	P4.0 (Input)
2	GND	GND	GND
3	TDO (Test Data Out)	P0.1 (Input)	P4.3 (Output)
4	V _{REF}	–	–
5	TMS (Test Mode Select)	P0.2 (Output)	P4.2 (Input)
6	nRESET	P0.4 (Open Drain)	nRESET (Input)
7	Keyed pin	–	–
8	+5V	–	–
9	TDI (Test Data In)	P0.3 (Output)	P4.1 (Input)
10	GND	GND	GND

No modifications were required for the slave MAXQ2000 EV kit. The 2 x 5 JTAG header was installed and connected as described above on the prototyping area of the master MAXQ2000 EV kit. Then the two EV kits were joined; a standard 2 x 5 JTAG cable (the type of JTAG generally used to connect the Serial-to-JTAG board to a MAXQ EV kit) was connected between the 2 x 5 JTAG header in the prototyping area on the master EV kit and the standard JTAG header (J4) on the slave EV kit. This JTAG 2 x 5 connector is included as part of the MAXQ2000 EV kit.

To simplify matters, no attempt was made to connect the power or reference voltage through the JTAG cable from the master EV kit to the slave EV kit. Instead, both EV kits were set up to run with the same V_{DDIO} voltage (approximately 3.6V). This setup ensured that the master and slave MAXQ2000s would operate with common I/O rail levels.

The slave EV kit also had the LCD daughterboard (MAXQ2000-K01) installed on header J3. Jumpers and DIP switch settings for both boards (along with the Serial-to-JTAG board) are listed in **Table 2**. **Note:** all jumpers not listed should be disconnected. **Figure 1** shows the final setup.

Table 2. Switch and Jumper Settings for Boards

Board	Switch or Jumper	Setting	Notes
Serial-to-JTAG Board	JH1	Connected	
	JH2	Connected	
	JH3	Connected	Supplies 5V power over JTAG cable
Master MAXQ2000 EV Kit	JU1	Pins 1 and 2 connected	Powers V_{DD} from 2.5V supply
	JU2	Pins 1 and 2 connected	Powers V_{DDIO} from 3.6V supply
	JU3	Pins 1 and 2 connected	Powers V_{LCD} from 3.6V supply
	JU11	Connected	Powers kit board from JTAG 5V supply
	DIP SW1	Switches #4 and #8 ON; all other switches OFF	Enables serial port 0 output to J5
	DIP SW3	All switches OFF	
	DIP SW6	All switches OFF	
Slave MAXQ2000 EV Kit	JU1	Pins 1 and 2 connected	Powers V_{DD} from 2.5V supply
	JU2	Pins 1 and 2 connected	Powers V_{DDIO} from 3.6V supply
	JU3	Pins 1 and 2 connected	Powers V_{LCD} from 3.6V supply
	DIP SW1	All switches OFF	
	DIP SW3	All switches OFF	
	DIP SW6	All switches OFF	

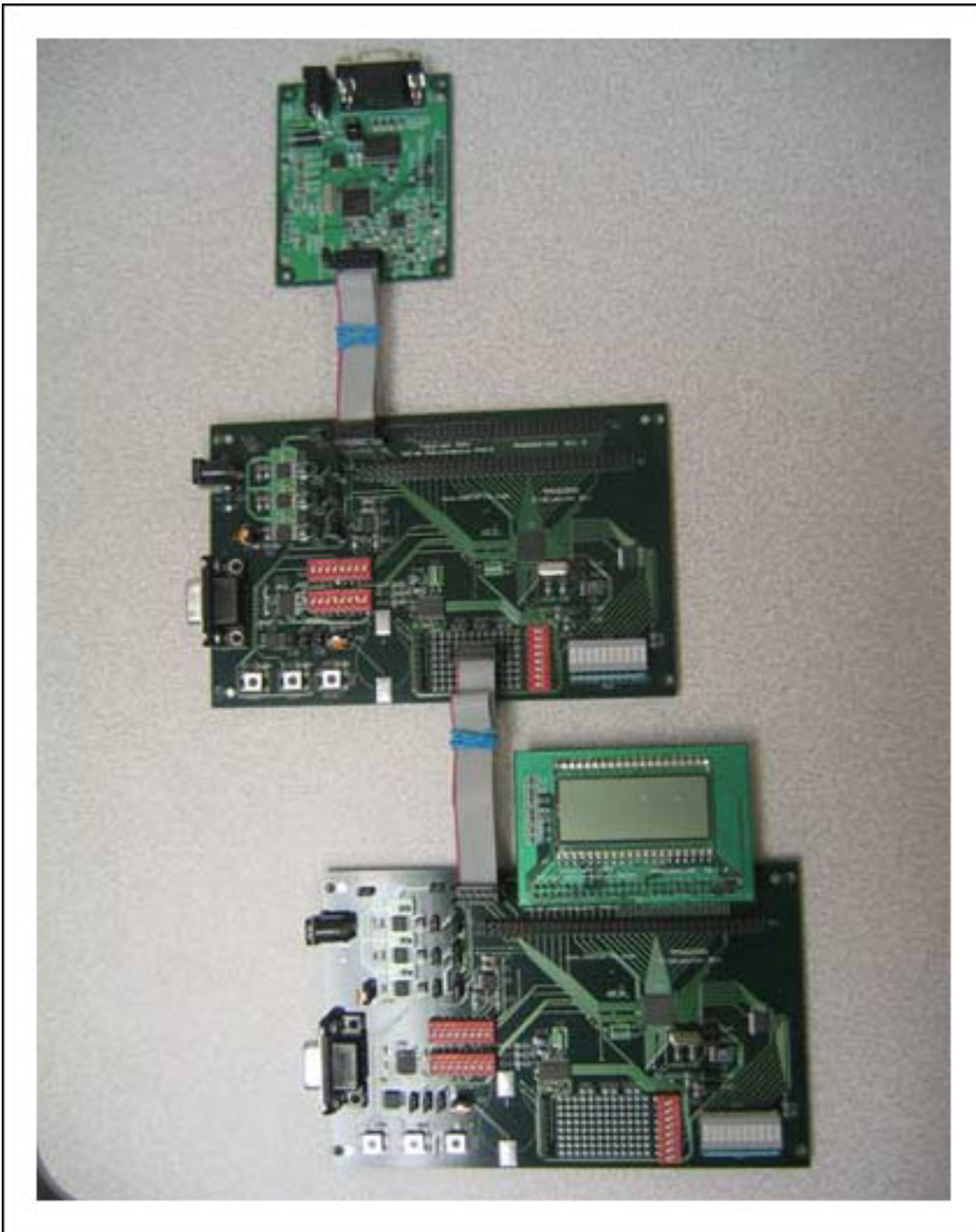


Figure 1. JTAG demo setup.

The MAXQ JTAG Interface

The JTAG interface on MAXQ microcontrollers consists of four signal lines which are used to shift information in and out of the Test Access Port (TAP) controller. This TAP controller, in turn, provides access to the MAXQ's bootloader and in-circuit debugging functions. (Note that implementing a debug master, while similar to implementing a bootloader master, is beyond the scope of this application note.) The four JTAG signal lines are listed in **Table 3**.

Table 3. JTAG Interface Signals

JTAG Signal	Signal Name	Direction (Master)	Direction (Slave)	Signal Description
TMS	Test Mode Select	Output	Input	This signal line, along with the TCK line, is used to shift the TAP controller from one operational state to the next.
TCK	Test Clock	Output	Input	This signal provides the clock for the JTAG interface. The JTAG clock is limited to a maximum of the slave's clock frequency divided by 8. For example, if the slave is running at a clock frequency of 8MHz, the JTAG clock at TCK cannot run any faster than 1MHz.
TDI	Test Data In	Output	Input	This signal carries data that is sent from the master to the slave.
TDO	Test Data Out	Input	Output	This signal carries data that is sent from the slave back to the master.

The nRESET pin is not, technically speaking, part of the JTAG interface. It is included on the JTAG cable to allow the JTAG master to reset the slave microcontroller. Resetting the slave microcontroller is a required step to enter bootloader mode, and can also be useful if JTAG communications are unexpectedly interrupted.

Since the JTAG interface is full-duplex, data is shifted in from the master to the slave on the TDI line at the same time that data is shifted out from the slave to the master on the TDO line. The slave samples incoming data (on both TDI and TMS) on the **rising edge** of TCK and drives outgoing data to the master on TDO on the **falling edge** of TCK. For both incoming and outgoing data, values are transferred with least significant bit first.

This application note provides only a brief overview of the JTAG interface and TAP controller in order to explain the operation of the example code. For a much more detailed discussion of these features, refer to the Test Access Port (TAP), In-Circuit Debug Mode, and In-System Programming sections of the [MAXQ Family User's Guide](#) which can be found on the Company website.

Communicating with the TAP Controller

The TAP controller is structured around a state machine, as shown below in **Figure 2**. There are sixteen discrete states included in the TAP state machine. Transition from one state to the next occurs on each rising edge of TCK based on the value of the TMS signal. For example, if the TAP controller is in the *Select-DR-Scan* state, and a rising edge occurs on TCK:

- If TMS = 1, the TAP controller will transition to the *Select-IR-Scan* state.
- If TMS = 0, the TAP controller will transition to the *Capture-DR* state.

In this manner, it is possible to clock the TAP controller through to any desired state. There are two things to note about the state diagram (Figure 2) below:

- Five '1' transitions (holding TMS high and clocking TCK for five full cycles) will always bring the state machine back to *Test-Logic-Reset*, regardless of the starting state. This means that if the current state of the TAP controller is unknown or if communication between the JTAG master and slave has been disrupted in some manner, it is always possible to bring the TAP controller back to a known state by clocking in five '1' transitions.
- It is possible to pause JTAG communication and remain in the *Run-Test-Idle*, *Pause-DR*, or *Pause-IR* states indefinitely without affecting the state of the TAP controller, even if the TCK clock continues running.

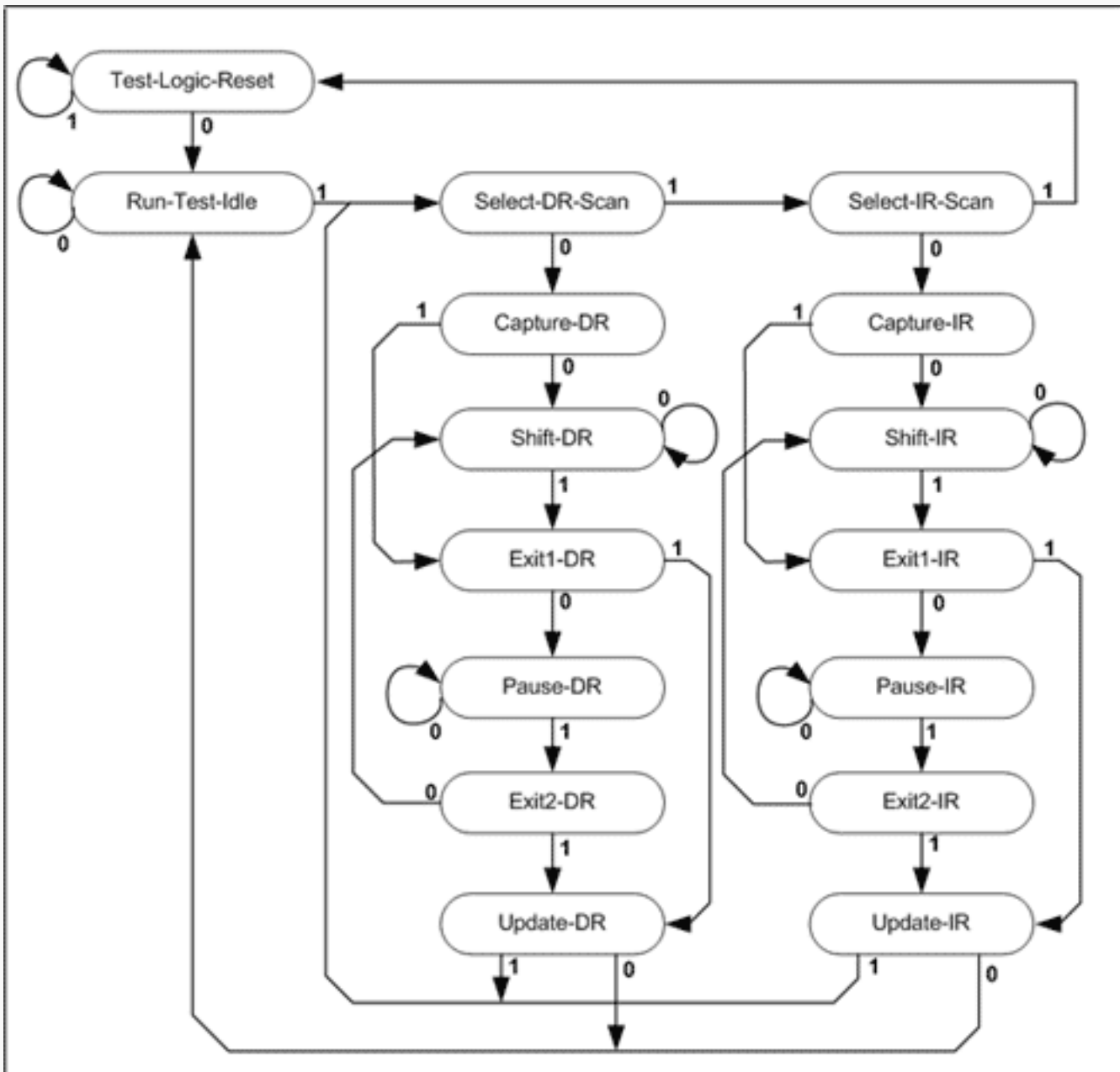


Figure 2. Test Access Port (TAP) state machine.

The TAP controller's state machine provides access to two control registers which, in turn, provide interfaces to the bootloader, the debug interface, and other functions.

- IR (Instruction Register) is always three bits in width. This register acts as an index register which, in turn, controls the function of DR (see below).
- DR (Data Register) is an access point to one of several registers within the TAP controller. The actual register accessed when bits are shifted into or out of DR depends on the current value of IR.

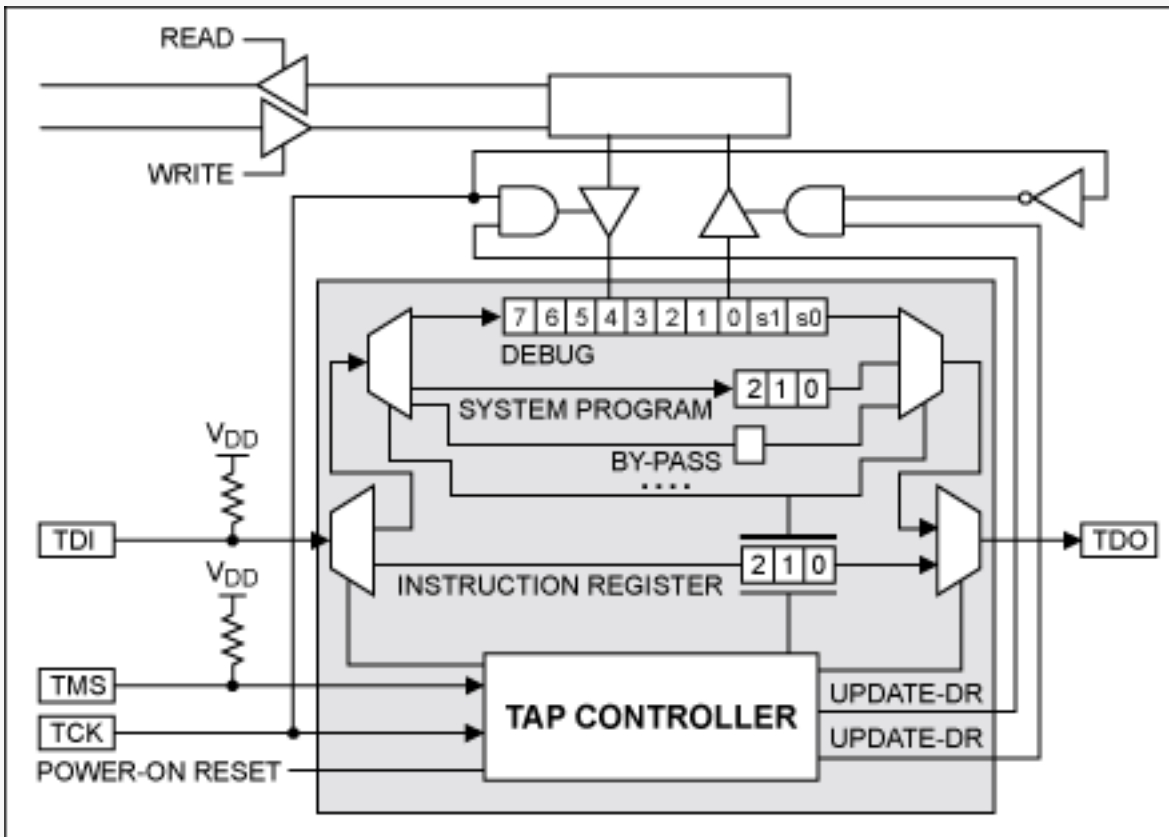


Figure 3. Register access in the TAP controller.

As shown in **Figure 3** (the DR points to one of three internal registers depending on the value of the IR.

- If **IR = 011b**, the TAP controller is in **Bypass** mode. In this mode (which is the default mode for the TAP controller), data shifted into DR (through TDI) is simply shifted back out again through TDO. No internal registers are altered by the data shifted through the TAP controller.
- Setting **IR = 100b** places the TAP controller in **System Programming** mode. In this mode, data shifted into DR is shifted into the 3-bit System Programming register. This register (also accessible by the MAXQ micro as bits [3:1] of the ICDF register) controls whether the MAXQ will enter normal program execution mode or bootloader mode following a reset. If bootloader mode is enabled, it also controls which interface (JTAG, serial, or SPI) will be used by the bootloader.
- Setting **IR = 010b** places the TAP controller in **Debug** mode. In this mode, data shifted into DR is shifted into an internal 10-bit debugging register which can be read by the bootloader. Data output by the bootloader is also shifted back out (along with two status bits) through this register and from there through TDO. This register is used to transfer data in both the bootloader and in-circuit debugging modes.

Of the three modes discussed above, the **Bypass** mode is a "no-operation" mode; it does not provide access to the bootloader functions in which we are interested. The **System Programming** mode, despite its name, is not actually used to communicate with the bootloader, only to enable access to it. Once the bootloader is active and communicating, this TAP mode ceases to be useful. The **Debug** mode, which allows access to the 10-bit input/output register, is the mode used to perform all communications with the bootloader. Once the bootloader has been enabled, this mode of the TAP controller is used exclusively until the loader session has been completed.

Controlling the JTAG Port and Reset Line

The four lines of the JTAG/TAP port on the slave MAXQ2000 (TMS, TCK, TDO, and TDI) and the nRESET line are each connected to one port pin on the master MAXQ2000. The first step in controlling the JTAG interface is to properly configure these lines.

```

#define TCK    PO0.0    ; Test Clock    - Master output
#define TDO    PI0.1    ; Test Data Out - Slave output, master input
#define TMS    PO0.2    ; Test Mode Sel - Master output
#define TDI    PO0.3    ; Test Data In  - Master output
#define RST    PD0.4    ; Reset        - Master open-drain output (on 1)

```

The four JTAG lines are operated in standard drive mode. From the master's perspective, TMS, TCK, and TDI will always be driven as outputs, while TDO (driven by the slave) will always be an input. The direction of the JTAG lines is fixed and not bidirectional. The nRESET line is a special case, and is configured to be an open-drain output from the master side. Normally, the slave will pull its own nRESET line high, so the master should only pull the line down (when resetting the slave) or release it completely (at all other times). There is no reason for the master to explicitly drive the slave's nRESET line high.

```

;=====
;=
;= initializeJTAG
;=
;= Sets up the port pins for the JTAG interface.
;=
;= Inputs    : None
;= Outputs   : None
;= Destroys  : None
;=
initializeJTAG:
    move    PD0.0, #1    ; TCK - master output
    move    PO0.0, #1    ; Drive high
    move    PD0.1, #0    ; TDO - master input
    move    PO0.1, #1    ; Weak pullup on
    move    PD0.2, #1    ; TMS - master output
    move    PO0.2, #1    ; Drive low
    move    PD0.3, #1    ; TDI - master output
    move    PO0.3, #1    ; Drive high
    move    PD0.4, #0    ; RST - open drain when 1, tristate when 0
    move    PO0.4, #0    ; Weak pullup off
    ret

```

After initializing the port pins, the `clock0` and `clock1` routines are used to clock static 0s and 1s on the TMS line to advance the TAP controller from one state to the next. The JTAG clock can be driven at any frequency, as long as the JTAG clock rate is kept below the maximum of 1/8 of the slave microcontroller's system clock rate. The master's system clock rate need not be considered here; it need not be matched to the slave's system clock rate in any way. The master can run faster or slower than the slave without causing JTAG communication problems.

Since the slave MAXQ2000 in this example has an 8MHz clock crystal installed, the master can drive the JTAG clock at up to 1MHz without problems. For this example, a JTAG clock rate of 100kHz will be sufficient.

```

#define JCLOCK    40    ; 100kHz : (((10us / (1/8MHz)) / 2)

;=====
;=
;= clock0
;=
;= Clocks a zero TMS bit into the JTAG interface.
;=
;= Inputs    : None
;= Outputs   : None
;= Destroys  : LC[0]

```

```

clock0:
    move    TMS, #0          ; Drive TMS low
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #1         ; Clock rising edge
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #0         ; Clock falling edge
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    ret

;=====
;=
;= clock1
;=
;= Clocks a one TMS bit into the JTAG interface.
;=
;= Inputs    : None
;= Outputs   : None
;= Destroys  : LC[0]

clock1:
    move    TMS, #1          ; Drive TMS high
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #1         ; Clock rising edge
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #0         ; Clock falling edge
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    ret

```

With these two routines in place, we can add a routine to initialize the TAP controller by forcing it back to the *Test-Logic-Reset* state. Note that the Test-Logic-Reset state, as its name suggests, performs a complete reset of the TAP logic, including the bits (SPE and PSS[1:0]) which determine whether or not the bootloader is enabled and which interface the bootloader is using. Therefore, once entering Bootloader mode, setting the TAP controller to Test-Logic-Reset will reset the device and exit the Bootloader mode. The JTAG routines used in the demo application (with the exception of `testLogicReset` itself) all assume that the TAP controller is in the *Run-Test-Idle* state at the start of the routine. During the JTAG communications routine, the TAP controller will be shifted through various states; at the end of the routine, the TAP controller will always be back at *Run-Test-Idle*.

```

;=====
;=
;= testLogicReset
;=    clock0, clock1
;=
;= Resets the JTAG/TAP controller to its starting state.
;=
;= Inputs    : None
;= Outputs   : None
;= Destroys  : LC[0]
;=

testLogicReset:
    call    clock0
    call    clock1

```

```

call    clock1
call    clock1
call    clock1
call    clock1
call    clock1
call    clock0          ; Brings us to Run-Test-Idle
ret

```

Writing to the TAP Instruction Register

Loading values into the TAP controller's IR is accomplished by traversing the TAP state machine down to the *Shift-IR* state and clocking in a new 3-bit value. The value is not actually copied from the shift register into the Instruction Register until the *Update-IR* state is entered.

As with all JTAG shift register operations, the current contents of the register are shifted out (by TDO) as the new value is shifted in. However, the value shifted out will always be fixed (001b); this is mandated by the JTAG standard for use in testing the functionality of the JTAG interface.

The procedure for shifting a bit in and out of either shift register (IR or DR) is identical; the TAP state machine must be in the *Shift-IR* or *Shift-DR* state, respectively. The input bit (at TDI) is sampled by the TAP controller on the rising edge of each TCK cycle, while the output bit (at TDO) is driven out on the falling edge of the TCK cycle.

```

;=====
;=
;=  shift
;=
;=  In a shift register state, clocks in a TDI bit and clocks out a TDO bit.
;=
;=  Inputs   : C - Bit to shift in to TDI.
;=  Outputs  : C - Bit shifted out from TDO.
;=  Destroys : PSW, LC[0]
;=

shift:
    jump    C, shift_bit1
shift_bit0:
    move    TDI, #0          ; Shift in zero bit
    jump    shift_bitEnd
shift_bit1:
    move    TDI, #1          ; Shift in one bit
    jump    shift_bitEnd
shift_bitEnd:
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #1          ; Rising edge, TDI is sampled
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    TCK, #0          ; Falling edge, TDO is driven out
    move    LC[0], #JCLOCK
    djnz    LC[0], $
    move    C, TDO           ; Latch TDO value
    ret

```

For each bit in the transfer, except the final bit, TMS must remain **low**. This is important so that, as each bit is shifted in and shifted out, the state machine remains in the *Shift-IR* or *Shift-DR* state. On the final bit cycle, TMS must be driven **high** so that, as the last bit is clocked in and out, the TAP controller will advance to the *Exit1-DR* or *Exit1-IR* state.

As an example, in **Table 4** the value **100b** (System Programming mode) is being shifted into the IR register by the JTAG master. The IR register starts out programmed to the bypass value **011b**; the TAP controller begins in the *Run-Test-Idle* state. As shown in **Figure 4** below, the bits are shifted into (and out of) the TAP controller beginning with the least significant bit and ending with the most significant bit. So on the first shift cycle, bit 0 of the new value is shifted in, and bit 0 of the old value is shifted out.

Table 4. Instruction Register Shift Example

TCK	TMS	TDI	TDO	TAP State	Shift Register			Instruction Register		
					bit 2	bit 1	bit 0	bit 2	bit 1	bit 0
0	1	x	x	Run-Test-Idle	x	x	x	0	1	1
1	1	x	x	Select-DR-Scan	x	x	x	0	1	1
0	1	x	x	Select-DR-Scan	x	x	x	0	1	1
1	1	x	x	Select-IR-Scan	x	x	x	0	1	1
0	0	x	x	Select-IR-Scan	x	x	x	0	1	1
1	0	x	x	Capture-IR	0	0	1	0	1	1
0	0	x	x	Capture-IR	0	0	1	0	1	1
1	0	x	x	Shift-IR	0	0	1	0	1	1
0	0	0	x	Shift-IR	0	0	1	0	1	1
1	0	0	x	Shift-IR	0	0	1	0	1	1
0	0	0	1	Shift-IR	0	0	0	0	1	1
1	0	0	1	Shift-IR	0	0	0	0	1	1
0	1	1	0	Shift-IR	0	0	0	0	1	1
1	1	1	0	Exit1-IR	0	0	0	0	1	1
0	1	x	0	Exit1-IR	1	0	0	0	1	1
1	1	x	0	Update-IR	1	0	0	1	0	0
0	0	x	x	Update-IR	1	0	0	1	0	0
1	0	x	x	Run-Test-Idle	x	x	x	1	0	0

The routine used to perform this operation, **shiftIR3**, is shown below.

```

;=====
;=
;= shiftIR3
;= clock0, clock1, shift
;=
;= Shifts a 3-bit value into the IR register.
;=
;= Inputs : A[0] - Low three bits contain value to shift into IR
;= Outputs : None
;= Destroys : AP, APC, A[0], PSW, LC[0]
;=

shiftIR3:
    move    APC, #80h           ; Acc => A[0], turn off auto inc/dec
    call    clock1              ; (Select DR Scan)
    call    clock1              ; (Select IR Scan)
    call    clock0              ; (Capture IR - loads 001b to shift register)
    call    clock0              ; (Shift IR)
    move    TMS, #0             ; Drive TMS low
    move    C, TDO              ; xxxxxx210 c = s
    rrc                                          ; sxxxxx21 c = 0

```

```

call    shift                ; Shift in IR bit 0
rrc     ; ssxxxxx2    c = 1
call    shift                ; Shift in IR bit 1
rrc     ; sssxxxxx    c = 2
move    TMS, #1             ; Drive TMS high for last bit
call    shift                ; Shift in IR bit 2 (Exit1 IR)
call    clock1              ; (Update IR)
call    clock0              ; (Run Test Idle)
ret

```

Writing to the TAP Data Register

Shifting values into and out of the DR of the TAP controller is performed in a similar manner to the load/unload of the IR. Normally, this will only be done when IR is set to one of two values: 100b (placing the TAP controller in System Programming mode) or 010b (placing the TAP controller in Debug mode).

When System Programming mode is active, loading and unloading the DR register operates as follows.

- The shift into and out of the DR register is three bits in width. All three bits represent valid data.
- The value transferred into the DR register is copied into the slave microcontroller's internal System Programming register when the *Update-DR* state is reached. These three bits are used as follows.
 - Bit 0 is accessible (read/write) by the slave microcontroller as register bit ICDF.1 (SPE), and is also known as the System Program Enable bit. This bit is examined by the Utility ROM following reset to determine if bootloader mode (SPE = 1) or normal program execution mode (SPE = 0) should be entered.
 - Bits 1 and 2 are accessible (read/write) by the slave microcontroller as register bits ICDF.2-3 (PSS0-PSS1), and are also known as the Programming Source Select bits. On microcontrollers that support more than one interface to the bootloader, such as the MAXQ2000, these bits are used to select which bootloader interface will be activated when SPE = 1. When SPE = 0 (normal program execution mode), the settings of these bits have no effect.
- The value transferred out of the DR register is the previous value of the System Programming register (latched into the shift register when the *Capture-DR* state is entered).

When Debug mode is active, loading and unloading the DR register operates as follows.

- The shift into and out of the DR register is 10 bits in width. For the data shifted out, all 10 bits represent valid data (eight data bits and two status bits). For the data shifted in, only the values of the eight data bits are used; the two status bits are not used.
- The high eight bits of the value shifted into the DR register are then unloaded and read by the bootloader (running from the Utility ROM) as part of a bootloader command.
- The 10-bit value shifted out of the DR register consists of an 8-bit value loaded by the bootloader (as part of a command output) and two bits of status information which are set by the TAP controller.

```

;=====
;=
;= shiftDR3
;=
;= Shifts a 3-bit value into the DR register. This operation should only be
;= performed when IR =100b (System Programming Mode).
;=
;= Inputs   : A[0] - Low 3 bits contain value to shift into SPB (PSS1:PSS0:SPE)
;= Outputs  : None
;= Destroys : AP, APC, A[0], PSW, LC[0]

shiftDR3:
    move    APC, #80h        ; Acc => A[0], turn off auto inc/dec
    call    clock1          ; (Select DR Scan)

```

```

call    clock0          ; (Capture DR)
call    clock0          ; (Shift DR)
move    TMS, #0         ; Drive TMS low
move    C, TDO          ; xxxxx210   c = s
rrc     ; sxxxxx21     c = 0
call    shift           ; Shift in DR bit 0
rrc     ; ssxxxxx2    c = 1
call    shift           ; Shift in DR bit 1
rrc     ; sssxxxxx    c = 2
move    TMS, #1         ; Drive TMS high for last bit
call    shift           ; Shift in DR bit 2 (Exit1 DR)
call    clock1          ; (Update DR)
call    clock0          ; (Run Test Idle)
ret

```

```

;=====
;=
;=  shiftDR
;=    clock0, clock1, shift
;=
;=  Shifts a 10-bit value into and out of the DR register.  This operation
;=  should only be performed when IR = 010b (Debug/Loader Mode).
;=
;=  Inputs   : A[0] - Byte value (input) to shift into DR
;=  Outputs  : A[0] - Byte value (output) shifted out of DR
;=            A[1] - Low two bits are status bits 1:0 shifted out of DR
;=            A[15] - Byte value shifted in, cached for use by shiftDR_next
;=  Destroys : AP, APC, PSW, LC[0]

```

shiftDR:

```

move    APC, #80h       ; Acc => A[0], turn off auto inc/dec
move    A[15], A[0]     ; Cache input byte value for use by shiftDR_next
sla2    ; Add two empty bits (for status)
call    clock1          ; (Select DR Scan)
call    clock0          ; (Capture DR)
call    clock0          ; (Shift DR)

move    TMS, #0         ; Drive TMS low
move    C, TDO          ; xxxxxxxx76543210   c = s
rrc
call    shift           ; Shift in DR bit 0
rrc
call    shift           ; Shift in DR bit 1
rrc
call    shift           ; Shift in DR bit 2
rrc
call    shift           ; Shift in DR bit 3
rrc
call    shift           ; Shift in DR bit 4
rrc
call    shift           ; Shift in DR bit 5
rrc
call    shift           ; Shift in DR bit 6
rrc
call    shift           ; Shift in DR bit 7
rrc
call    shift           ; Shift in DR bit 8
rrc
move    TMS, #1         ; Drive TMS high for last bit
call    shift           ; Shift in DR bit 9 (Exit1 DR)

```

```

call    clock1          ; (Update DR)
call    clock0          ; (Run Test Idle)

push    Acc             ; sddd dddd 10xx xxxx
sra4    ; ssss sddd dddd 10xx
sra2    ; ssss sssd dddd dd10
and     #0003h         ; ---- ---- ---- --10
move    A[1], Acc      ; Return status bits only in A[1]
pop     Acc
and     #0FF00h
xch                    ; Return data bits only in A[0]
ret

```

Entering JTAG Bootloader Mode

The following steps are required to put the MAXQ2000 into JTAG bootloader mode.

1. Initialize the TAP controller, resetting it to the *Test-Logic-Reset* state.
2. Set the Instruction Register (IR) to 100b to enable System Programming mode.
3. Set the Data Register (DR) to 001b. This sets the SPE (System Programming Enable) bit to 1 to enable the bootloader, and sets the PSS[1:0] (Programming Source Select) bits to 00b to select the JTAG interface.
4. Hold nRESET low to reset the MAXQ2000.
5. Release nRESET. This causes the MAXQ2000 to vector to the standard reset point in the Utility ROM (8000h). The Utility ROM code will then examine the values of the SPE and PSS bits and activate the JTAG bootloader accordingly. At this point, the bootloader is running and ready to accept JTAG commands.
6. Set the Instruction Register (IR) to 010b to enable Debug mode. This is the mode used to communicate with either the JTAG bootloader or the debug engine; we will be using this mode to communicate with the bootloader in this case.
7. Begin sending commands to the JTAG bootloader by shifting 10-bit values through DR.

```

#define IR_DEBUG          010b      ; Debug Mode
#define IR_BYPASS        011b      ; Bypass Mode (default)
#define IR_SYSTEM_PROG   100b      ; System Programming Mode (activate loader)

; System Programming Register settings

#define SP_EXECUTE       000b      ; Bootloader disabled
#define SP_LOAD_JTAG     001b      ; Activate JTAG bootloader
#define SP_LOAD_UART     011b      ; Activate UART bootloader
#define SP_LOAD_SPI      101b      ; Activate SPI bootloader (invalid on 2000)
#define SP_RESERVED     111b      ; Reserved value

...

call    initializeJTAG    ; Set up port pins for JTAG
call    testLogicReset   ; Reset JTAG port (ending state: Run-Test-Idle)

move    Acc, #IR_SYSTEM_PROG
call    shiftIR3         ; Load the System Programming instruction into IR

move    Acc, #SP_LOAD_JTAG
call    shiftDR3         ; Enable the bootloader in JTAG interface mode

move    RST, #1          ; Drive nRESET low
move    Acc, #100        ; Delay for 100ms
call    delayMS

```

```

call    clock0                ; Remain in Run-Test-Idle

move    RST, #0               ; Release nRESET
move    Acc, #100             ; Delay for 100ms
call    delayMS

move    Acc, #IR_DEBUG
call    shiftIR3              ; Enable access to the 10-bit debug shift register

;;; Bootloader commands may now be shifted through the DR register

call    waitForPrompt         ; Verify that the bootloader is responding

...

```

Communicating with the Bootloader

Once the bootloader is running, the bootloader code in the Utility ROM reads command codes from an internal register loaded from the DR shift register. The Utility ROM also writes result data to another internal register that can be shifted out through the DR register by the JTAG master. In this way, the bootloader code and the JTAG master can exchange information.

The bootloader code and the JTAG master are, however, not synchronized. As long as the JTAG clock's rate remains below the maximum of 1/8 of the MAXQ system clock, the exact values of the two clocks do not matter for communication purposes because JTAG is a synchronous interface. However, the MAXQ system clock rate and the nature of the bootloader commands received will determine how long a delay occurs between when the bootloader receives a command and sends a reply. For example, a MAXQ2000 running at 1MHz will take longer to respond to a given bootloader command than a MAXQ2000 running at 10MHz, even though both microcontrollers can communicate using a JTAG clock of 100kHz. Commands which simply read and return information, such as commands to return the ROM banner ID or read from program memory, also take less time than commands which include operations such as programming flash memory.

The data transferred through the JTAG interface is not buffered. If another 10-bit command is sent before the previous command has been read, the previous command's result is lost. The ROM code and TAP controller ensure that no additional data will be sent out by the bootloader before the previous data has been unloaded by the JTAG master, but synchronization in the reverse direction is needed as well. The JTAG master needs a way to tell if the previous byte that it shifted into DR has been read by the bootloader. In that way the JTAG master can tell when it is time to send the next byte. The method for knowing this derives from the two additional status bits sent out by the TAP controller with each 8-bit byte output of the bootloader.

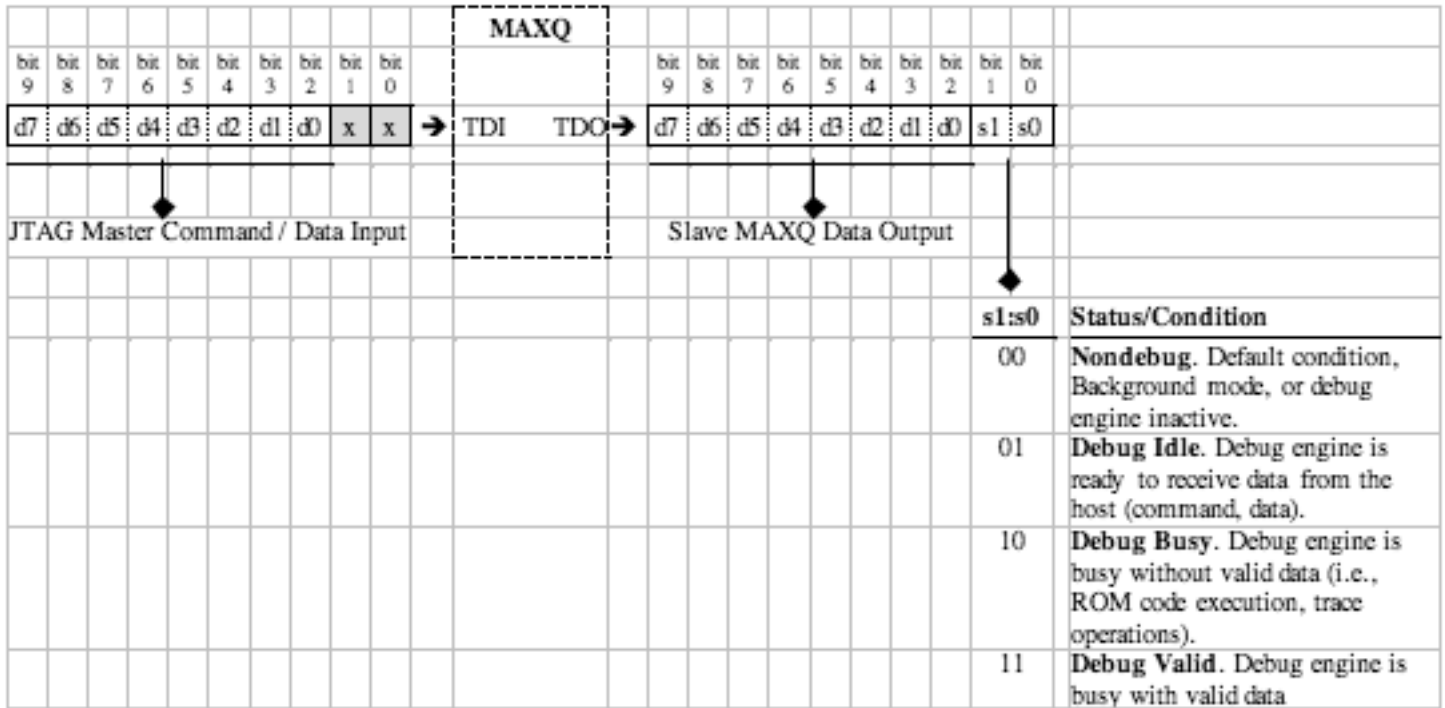


Figure 4. Shifting Data and Status Bits Through DR.

As shown above in Figure 4, the data shifted into and out of DR in TAP Debug mode consists of eight data bits (bits 2 through 9) and two status bits (bits 0 and 1). When the JTAG master shifts data in, only the eight data bits are used. The two status bits (which still must be shifted in) are not used by the TAP controller and can be set to zero or any other value.

In the 10-bit value shifted out from the TAP controller, the two status bits provide information about the state of the bootloader or the debug engine. When the bootloader is running, only the last two states (Debug Busy or Debug Valid) will normally be encountered, as the other two status values do not occur in bootloader mode. (See Status/Condition in Figure 4.)

- If the status bits are set to the Debug Busy (10b) value, the bootloader has **not yet read** the previous value that was shifted into DR by the JTAG master. This also means that the 8-bit data value shifted out is meaningless, since the bootloader has not yet completed the command. When this value is received, the JTAG master must then reload DR with the **previously transmitted** byte value, so that the bootloader can access it. The state sequence to perform this operation correctly is as follows.
 - After shifting the last bit into DR in the *Shift-DR* state and transitioning to *Exit1-DR*, the two status bits should be checked by the JTAG master.
 - If the Debug Busy value was received, transition to the *Pause-DR* state and from there to *Exit2-DR*, then back to *Shift-DR* again. Reload the **previous** DR value (not the value that was shifted in this time), and then go through *Exit1-DR* and *Update-DR*, disregarding the value of the status bits on the second pass through.
 - Delay for a short period of time (depending on the command being executed) and retry the byte transfer.
- If the status bits are set to the Debug Valid (11b) value, the bootloader has read the last byte shifted in and has loaded a reply byte. The 8-bit value shifted out contains valid data.

When executing a bootloader command sequence, the `shiftDR` and `shiftDR_next` routines perform this synchronization automatically. The `shiftDR` routine should be called for the first byte in a sequence; `shiftDR_next` is called for subsequent bytes. The `shiftDR_next` routine operates identically to `shiftDR` except that it checks the two status bits and retransmits the previous DR byte as described above, if necessary. The `sendCommand` routine ties all this together and calls `shiftDR` and `shiftDR_next` to transmit a command sequence, delaying and resending bytes as needed until the command has completed.

```

;=====
;=
;= sendCommand
;=
;= Transmits a loader command by shifting bytes through DR.
;=
;= Inputs   : DP[0] - Points to area of RAM which stores input bytes
;=           for command and which will be filled with output.
;=           LC[1] - Number of bytes to transmit/receive
;= Outputs  : C - Set on JTAG communication error
;= Destroys : AP, APC, A[0], A[1], A[2], A[15], PSW, LC[0]
;=

sendCommand:
    move    APC, #80h          ; Acc => A[0], turn off auto inc/dec
    push   LC[1]
    call   waitForPrompt
    pop    LC[1]
    jump   C, sendCommand_fail

    move    Acc, @DP[0]        ; Read first byte to transmit
    call   shiftDR
    push   Acc
    move    Acc, A[1]
    cmp    #3                  ; Should be valid status since we had a prompt
    pop    Acc
    jump   NE, sendCommand_fail

    move    @DP[0], Acc        ; Store first received byte
    move    NUL, @DP[0]++      ; Increment data pointer
    djnz   LC[1], sendCommand_loop
    jump   sendCommand_pass

sendCommand_loop:
    move    A[2], #10         ; Number of retries allowed
sendCommand_retry:
    move    Acc, @DP[0]        ; Get next byte to transmit
    call   shiftDR_next
    push   Acc
    move    Acc, A[1]
    cmp    #3
    pop    Acc
    jump   NE, sendCommand_stall

    move    @DP[0], Acc        ; Store received byte
    move    NUL, @DP[0]++      ; Increment data pointer
    djnz   LC[1], sendCommand_loop
    jump   sendCommand_pass

sendCommand_stall:
    move    LC[0], #8000      ; About a millisecond
    djnz   LC[0], $
    move    Acc, A[2]
    sub    #1
    jump   NZ, sendCommand_retry
    jump   sendCommand_fail

```

Functions Provided by the Bootloader

Bootloader functions on MAXQ microcontrollers generally follow a shared pattern, which means that many of the commands and status codes are identical among the devices. Different devices can implement different subsets of the bootloader command set, depending on the structure of their internal program memory and other requirements. For specific details, consult the *User's Guide Supplement* for the MAXQ microcontroller that you are using. In this case, we will refer to the bootloader commands in the "In-System Programming" section of the *MAXQ2000 User's Guide Supplement*.

As with other MAXQ bootloaders, the commands provided by the MAXQ2000 bootloader are divided into command families from 0 to 15. Each command begins with a command byte, which is a combination of the command family (top four bits) and a number specific to the command (low four bits), as shown in **Table 5**. As a general rule, Family 0 commands, which are informational in nature, are implemented by all devices; other command families are optional. To determine the command families supported by a specific MAXQ device, please refer to the device's documentation. The Family 0 command 05h (Get Supported Commands) returns a bitmask indicating which other command families are supported by that bootloader.

Bootloader command families supported by the MAXQ2000 follow.

- Family 0—Information and Status. The commands in this family can be used to obtain basic information about the MAXQ device, including the identity and version of the ROM/bootloader, the result (status code) from the most recent command, and the sizes of program and data memory. This family also includes the Master Erase command, which clears all program and data memory on the device.
- Family 1—Load Variable Length. The commands in this family can be used to load program (flash) or data (RAM) memory.
- Family 2—Dump Variable Length. The commands in this family can be used to read the contents of program or data memory.
- Family 3—CRC Variable Length. The commands in this family can be used to obtain a CRC-16 value calculated over a specified range of program or data memory.
- Family 4—Verify Variable Length. The commands in this family can be used to verify whether or not a specified range of program or data memory matches data provided by the JTAG master.
- Family 5—Load and Verify Variable Length. The commands in this family combine the functionality of the Load and Verify commands into one command.
- Family 6—Erase Variable Length. On the MAXQ2000, this command can be used to clear a specified area of data RAM to zero.
- Family 7—Erase Fixed Length. On the MAXQ2000, this command can be used to erase individual pages of flash program memory, instead of erasing all flash memory at once with the Master Erase command.

Once the TAP controller on a MAXQ microcontroller has been used to place that microcontroller in JTAG loader mode (a process which is the same for all MAXQ microcontrollers), the first step is to determine whether the loader is responding. The quickest way to do this is to transmit the bootloader command code 00h (No Operation) repeatedly. After each bootloader command has completed, the bootloader responds with a prompt ('>' or 3Eh) byte. If transmitting a 00h results in a 3Eh reply, the bootloader is running and is ready to accept commands.

Table 5. Generic MAXQ Bootloader Command Families

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Code	Family/Command
0	0	0	0	x	x	x	x	0 x h	Family 0—Informational Commands
0	0	0	0	0	0	0	0	00h	No Operation
				0	0	0	1	01h	Exit Loader
				0	0	1	0	02h	Master Erase
				0	0	1	1	03h	Password Match
				0	1	0	0	04h	Get Status
				0	1	0	1	05h	Get Supported Commands
				0	1	1	0	06h	Get Code Memory Size
				0	1	1	1	07h	Get Data Memory Size
				1	0	0	0	08h	Get Loader Version
				1	0	0	1	09h	Get Utility ROM Version
0	0	0	0	1	0	1	0	0Ah	Set Word/Byte Access Mode
				1	1	0	1	0Dh	Get ID Information
0	0	0	1	x	x	x	x	1 x h	Family 1—Variable-Length Load
0	0	0	1	0	0	0	0	10h	Load Code Variable Length
0	0	0	1	0	0	0	1	11h	Load Data Variable Length
0	0	1	0	x	x	x	x	2 x h	Family 2—Variable-Length Dump
0	0	1	0	0	0	0	0	20h	Dump Code Variable Length
0	0	1	0	0	0	0	1	21h	Dump Data Variable Length
0	0	1	1	x	x	x	x	3 x h	Family 3—Variable-Length CRC
0	0	1	1	0	0	0	0	30h	CRC Code Variable Length
0	0	1	1	0	0	0	1	31h	CRC Data Variable Length
0	1	0	0	x	x	x	x	4 x h	Family 4—Variable-Length Verify
0	1	0	0	0	0	0	0	40h	Verify Code Variable Length
0	1	0	0	0	0	0	1	41h	Verify Data Variable Length
0	1	0	1	x	x	x	x	5 x h	Family 5—Variable-Length Load and Verify
0	1	0	1	0	0	0	0	50h	Load/Verify Code Variable Length
0	1	0	1	0	0	0	1	51h	Load/Verify Data Variable Length
0	1	1	0	x	x	x	x	6 x h	Family 6—Variable-Length Erase
0	1	1	0	0	0	0	0	60h	Erase Code Variable Length
0	1	1	0	0	0	0	1	61h	Erase Data Variable Length
0	1	1	1	x	x	x	x	7 x h	Family 7—Reserved (for expansion)
1	0	0	0	x	x	x	x	8 x h	Family 8—Reserved (for expansion)
1	0	0	1	x	x	x	x	9 x h	Family 9—Fixed-Length Load
1	0	0	1	0	0	0	0	90h	Load Code Fixed Length
1	0	0	1	0	0	0	1	91h	Load Data Fixed Length
1	0	1	0	x	x	x	x	A x h	Family A —Fixed-Length Dump
1	0	1	0	0	0	0	0	A0h	Dump Code Fixed Length
1	0	1	0	0	0	0	1	A1h	Dump Data Fixed Length
1	0	1	1	x	x	x	x	B x h	Family B—Fixed-Length CRC
1	0	1	1	0	0	0	0	B0h	CRC Code Fixed Length

1	0	1	1	0	0	0	1	B1h	CRC Data Fixed Length
1	1	0	0	x	x	x	x	C x h	Family C—Fixed-Length Verify
1	1	0	0	0	0	0	0	C0h	Verify Code Fixed Length
1	1	0	0	0	0	0	1	C1h	Verify Data Fixed Length
1	1	0	1	x	x	x	x	D x h	Family D—Fixed-Length Load and Verify
1	1	0	1	0	0	0	0	D0h	Load/Verify Code Fixed Length
1	1	0	1	0	0	0	1	D1h	Load/Verify Data Fixed Length
1	1	1	0	x	x	x	x	E x h	Family E —Fixed-Length Erase
1	1	1	0	0	0	0	0	E0h	Erase Code Fixed Length
1	1	1	0	0	0	0	1	E1h	Erase Data Fixed Length
1	1	1	1	x	x	x	x	F x h	Family F—Reserved (device specific)

Identifying the JTAG Slave Microcontroller

After the bootloader is up and running, the next step is to identify the slave MAXQ microcontroller. The Family 0 command 0Dh (Get ID Information) returns a variable-length ASCII string (zero terminated) identifying the device and the utility ROM version.

Our example program obtains this information and then prints it out to the serial port as part of the demo. The routine used to do this is `getBanner`.

```

;=====
;=
;=  getBanner
;=      waitForPrompt
;=      shiftDR
;=      clock0, clock1, shift
;=
;=  Executes command 0Dh to retrieve the ROM (device ID) banner and prints
;=  the banner text out over the serial port.
;=
;=  Inputs   : None
;=  Outputs  : C - Set on JTAG communication error
;=  Destroys : AP, APC, Acc, PSW, LC[0]
;=

getBanner:
    call    waitForPrompt
    jump    C, getBanner_fail

    move    Acc, #CMD_GET_ROM_BANNER
    call    shiftDR
    move    Acc, #00h
    call    shiftDR

getBanner_loop:
    move    Acc, #00h
    call    shiftDR
    cmp     #0FFh                ; The banner is ASCII, so receiving this character
                                ; most likely indicates that the JTAG lines are
                                ; floating high and that no device is connected

    jump   E, getBanner_fail
    jump   Z, getBanner_done
    call   txChar

```

```

    jump    getBanner_loop

getBanner_done:
    call    txNewline
    jump    getBanner_pass

getBanner_fail:
    move    C, #1
    ret

getBanner_pass:
    move    C, #0
    ret

```

Erasing Program Memory

Flash program memory, which is included on the MAXQ2000, must be erased (set to 0FFFFh) before it can be programmed. The JTAG demo application erases the flash memory by sending the 02h (Master Erase) bootloader command, instructing the bootloader to erase all program and data memory. This command also clears the password lock bit, thereby enabling all supported command families.

Depending on the MAXQ device, this 02h (Master Erase) command can require a few seconds to complete. As this is a single-byte command, the easiest way to determine when it is finished is to send the No Operation (00h) command followed by a 1 millisecond delay continuously until the bootloader returns a 3Eh, indicating command complete. This methodology is shown in the **masterErase** routine below.

```

;=====
;=
;=  masterErase
;=
;=  Executes command 02h (Master Erase) to clear all program and data memory.
;=
;=  Inputs    : None
;=  Outputs   : C - Set on JTAG communication error
;=  Destroys  : Acc, PSW, LC[0], LC[1]
;=
masterErase:
    call    waitForPrompt
    jump    C, masterErase_fail

    move    Acc, #CMD_MASTER_ERASE
    call    shiftDR
    move    Acc, #00h
    call    shiftDR

    move    LC[1], #5000    ; Number of retries before returning an error
masterErase_loop:
    move    Acc, #CMD_NOP
    call    shiftDR
    cmp     #3Eh
    jump    E, masterErase_pass
    move    LC[0], #8000    ; Delay for about a millisecond
    djnz   LC[0], $
    djnz   LC[1], masterErase_loop

masterErase_pass:
    move    C, #0
    ret

```

```

masterErase_fail:
    move    C, #1
    ret

```

Retrieving Status Information

After MasterErase (or virtually any other bootloader command) has finished, the 04h (Get Status) command can be used to determine whether or not the command completed successfully. The Get Status command returns two bytes of data: one byte contains status flags (indicating password lock set/unset, word/byte mode active, and other information) and the second byte is a single-byte status code.

The status code is always 00h (No Error) if the last command completed successfully. A nonzero status code indicates an error. An exhaustive list of status codes is available in the *MAXQ2000 User's Guide Supplement*; a few common error codes are listed here.

- 01h/02h—Family Not Supported/Invalid Command. These error codes usually indicate a communication glitch or alignment error with the JTAG master. If the JTAG master sends more (or fewer) bytes than the bootloader expects for a given command code, the bootloader will interpret one of the data bytes as the start of a new command.
- 03h—No Password Match. This error code usually indicates that the JTAG master tried to use a command which was password-protected (which generally includes any command not in Family 0) without first clearing the password lock. This error happens if the JTAG master accesses a part which already has code loaded into word addresses 0010h – 001Fh. In this case, the part must either be Master Erased or the Password Match command (03h) must be used to unlock the part.
- 05h—Verify Failed. The verification step failed on a Load/Verify or a Verify command. This error can occur after a communications glitch, or also because an attempt was made to reprogram previously programmed flash memory without first erasing it.

Loading Code into Program Memory

To demonstrate the JTAG bootloader's functionality, the demo application needs to load code into the slave MAXQ2000 over the JTAG connection. In this example, the code loaded will be a simple routine to start the LCD controller and show a 4-digit number on the LCD display.

The code loaded into the slave MAXQ2000 is based on the following simple demo project.

```

org 0
    ljump    main

org 20h

main:
    move    LCRA, #03E0h        ; xxx0001111100000
                                ;    00          - DUTY : Static
                                ;    0111         - FRM  : Frame freq
                                ;        1         - LCCS : HFClk / 128
                                ;        1         - LRIG : Ground VADJ
                                ;        00000    - LRA  : RADJ = Min

    move    LCFG, #0F3h        ; 1111xx11
                                ; 1111         - PCF  : All segments enabled
                                ;    1         - OPM  : Normal operation
                                ;    1         - DPE  : Display enabled

    move    LCD0, #LCD_CHAR_0
    move    LCD1, #LCD_CHAR_0
    move    LCD2, #LCD_CHAR_0

```

```

move    LCD3, #LCD_CHAR_2
move    LCD4, #00h

sjump   $

```

However, since we are loading the code bytes from a demo application instead of a hard-coded hex file (as would be the case when loading code with MTK or MAX-IDE), the demo application will modify the application loaded based on user input.

Before the code is loaded, the JTAG communications demo application first instructs the user to enter a 4-digit decimal number and then modifies the application being loaded as follows.

- The number displayed on the LCD is the 4-digit number entered by the user.
- The first four bytes of the password area (starting at word address 010h) are programmed to the ASCII values of the four characters entered by the user.

The application code is loaded by the demo using three calls to the bootloader command 10h (Load Code, Variable Length). The parameters to this command include: the number of bytes to load (must be an even number when loading code memory to ensure word alignment); the starting address; and of course, the data itself. MAXQ memory is little-endian, therefore the application must send the least significant byte of each program word first.

```

;;;
;;; First load - LJUMP 0020h at start of program memory
;;;

move    DP[0], #0
move    @DP[0], #CMD_LOAD_CODE_VARIABLE
move    @++DP[0], #4           ; Length - 4 bytes
move    @++DP[0], #00h        ; AddrL (byte address 0000h)
move    @++DP[0], #00h        ; AddrH

move    @++DP[0], #000h        ; 00 0B 20 0C - ljump 0020h
move    @++DP[0], #00Bh
move    @++DP[0], #020h
move    @++DP[0], #00Ch
move    @++DP[0], #000h        ; Padding
move    @++DP[0], #000h        ; Padding
move    @++DP[0], #55h
move    LC[1], DP[0]
move    DP[0], #0

nop
call    sendCommand
nop
jump    C, main_failJTAG
call    getStatus
jump    C, main_failJTAG
move    Acc, A[3]             ; Check that loader status is 00h (no error)
jump    NZ, main_failStatus

```

The remaining two blocks of code memory are loaded in a similar fashion.

Verifying Code in Program Memory

Once the code is loaded, there are several methods to verify that it loaded correctly.

- Instead of using the Load Code Variable Length command (10h), use the Load and Verify Code Variable Length command (50h). This latter command combines the code load operation with a verification step.
- Alternatively, the verification can be performed separately by calling the Verify Code Variable Length command (40h).
- Instead of having the bootloader verify the loaded code, the JTAG host can verify the loaded values directly by using the Dump Code Variable Length (20h) to verify that the loaded code matches the data sent.

Dumping Code from Program Memory

After all code is loaded, the final step in the JTAG demo is to read all the code back out in a single block. This read is done by sending the Dump Code Variable Length (20h) command to the bootloader. The parameters for this command are the address to begin dumping (reading) from, and the number of bytes to dump. Note that, as with all JTAG bootloader commands, a byte must be sent for every byte read back. Consequently, there are a large number of zero pad bytes needed by this command, one for each byte that will be read from program memory.

```

;;;
;;; Dump program code
;;;

move    DP[0], #str_dumpCodeVariable
call    txString

move    DP[0], #0
move    @DP[0], #CMD_DUMP_CODE_VARIABLE
move    @++DP[0], #1           ; Indicates single byte length (<256 bytes)
move    @++DP[0], #00h        ; AddrL (byte address 0000h)
move    @++DP[0], #00h        ; AddrH
move    @++DP[0], #128        ; Length - 128 bytes

move    LC[1], #128
main_loop1:
move    @++DP[0], #00h
djnz    LC[1], main_loop1

move    @++DP[0], #000h        ; Padding
move    @++DP[0], #000h        ; Padding
move    @++DP[0], #55h
move    LC[1], DP[0]
move    DP[0], #0

nop
call    sendCommand
nop
jump    C, main_failJTAG
call    getStatus
jump    C, main_failJTAG
move    Acc, A[3]             ; Check that loader status is 00h (no error)
jump    NZ, main_failStatus

```

The JTAG demo application outputs these code bytes over the serial port in hex format after they are read.

Exiting the Bootloader

The single-byte Family 0 command 01h (Exit Loader) causes the bootloader to complete operations and exit as follows.

- The bootloader initiates an internal reset which clears the SPE and PSS bits to zero and resets the microcontroller.
- The microcontroller exits reset and begins execution in the Utility ROM at 8000h.
- With SPE now cleared to zero, the Utility ROM code causes execution to jump to the start of user application code at address 0000h.

Since the Exit Loader command automatically clears the SPE and PSS bits, and since the subsequent reset clears the TAP controller back to Bypass mode, there is no need for the JTAG master to perform either of these actions.

```
;;;
;;; Exit loader mode and allow program code to execute
;;;

call    waitForPrompt
move    Acc, #CMD_EXIT_LOADER
call    shiftDR
move    Acc, #00h
call    shiftDR
move    Acc, #00h
call    shiftDR
move    Acc, #00h
call    shiftDR
```

Running the Demo

The following hardware and software components are required to run the JTAG bootloader demo.

Hardware

- Two MAXQ2000 Evaluation Kit boards ([MAXQ2000-K00](#) REV B); one EV kit will act as the master MAXQ2000 kit and the other will serve as the slave MAXQ2000 kit.
- MAXQ2000 LCD Daughterboard (MAXQ2000-K01 REV B)
- Two 2 x 5 JTAG interface cables (included with the MAXQ2000 EV kit)
- Serial-to-JTAG Interface Board (MAXQJTAG-001 REV B)
- DB9 straight-through serial cable
- Two 5V regulated ($\pm 5\%$) DC wall supplies, center post positive, CUI Inc. DPR050030-P6 or equivalent
- Two HC49US 8.00MHz crystals
- 2 x 5 0.100inch pin header

Software

- JTAG demo software package
 - http://files.dalsemi.com/microcontroller/app_note_software/an4012_sw.zip
- MAX-IDE Development Environment for MAXQ
 - http://files.dalsemi.com/microcontroller/maxq/dev_tool_software/MAX-IDE/MAX-IDE.zip
- Microcontroller Tool Kit (MTK) for MAXQ
 - http://files.dalsemi.com/microcontroller/dev_tool_software/mtk/

Setup Instructions

1. If MAX-IDE is not already installed, download and install according to the documentation included with the MAXQ2000 EV kit.
2. If MTK is not already installed, download and install according to the documentation included with the MAXQ2000 EV kit.
3. Connect the LCD daughterboard to header J3 on the slave MAXQ2000 Kit, as shown in Figure 1 above. The LCD daughterboard should be hanging off the top side of the MAXQ2000 kit board.
4. Install the 8.00MHz crystals on both MAXQ2000 EV kit boards (at Y1).
5. Install the 2 x 5 JTAG header in the prototyping area of the master MAXQ2000 kit, with pins connected to the master MAXQ2000 pins, as listed in Table 1.
6. Configure the jumpers and DIP switches on the Serial-to-JTAG board and both MAXQ2000 EV kit boards, as listed in Table 2.
7. Connect the DB9 serial cable from the COM1 port on the PC to J1 on the Serial-to-JTAG board.
8. Connect the first 5V power supply to J2 on the Serial-to-JTAG board.
9. Connect the second 5V power supply to J1 on the slave MAXQ2000 EV kit board.
10. Connect the first JTAG cable from P2 on the Serial-to-JTAG board to J4 on the master MAXQ2000 kit. The red wire should go to pin 1 on both JTAG headers.
11. Connect the second JTAG cable from the prototype area JTAG header on the master MAXQ2000 kit to J4 on the slave MAXQ2000 kit. The red wire should go to pin 1 on both JTAG headers.

Compiling and Loading the JTAG Demo

1. Download the JTAG demo software package and unzip it into a working directory.
2. Start MAX-IDE.
3. Turn on power to both 5V power supplies.
4. Select **Project → Open Project** from the menu, and select the **maxqjtag.prj** project file to open it.
5. Select **Debug → Make** from the menu. A "Build Successful" message should appear.
6. Select **Debug → Run** from the menu. A series of "Loading" messages should appear, followed by "Done".
7. Select **Debug → Stop** from the menu.
8. Close MAX-IDE.
9. Turn off power.

Running the JTAG Demo

1. With power off, disconnect the DB9 serial cable from the Serial-to-JTAG board.
2. Connect the DB9 cable to J5 on the master MAXQ2000 kit board.
3. Start MTK. Select "Dumb Terminal" in the startup dialog box.
4. Select **Options → Configure Serial Port** from the menu. In the dialog box, set Port to **COM1** and Speed to **9600** baud.
5. Select **Target → Open COM1 at 9600 baud** from the menu.
6. Turn power on.

If the application is loaded and everything is connected correctly, the following text (**Figure 5**) will be output over the serial port.

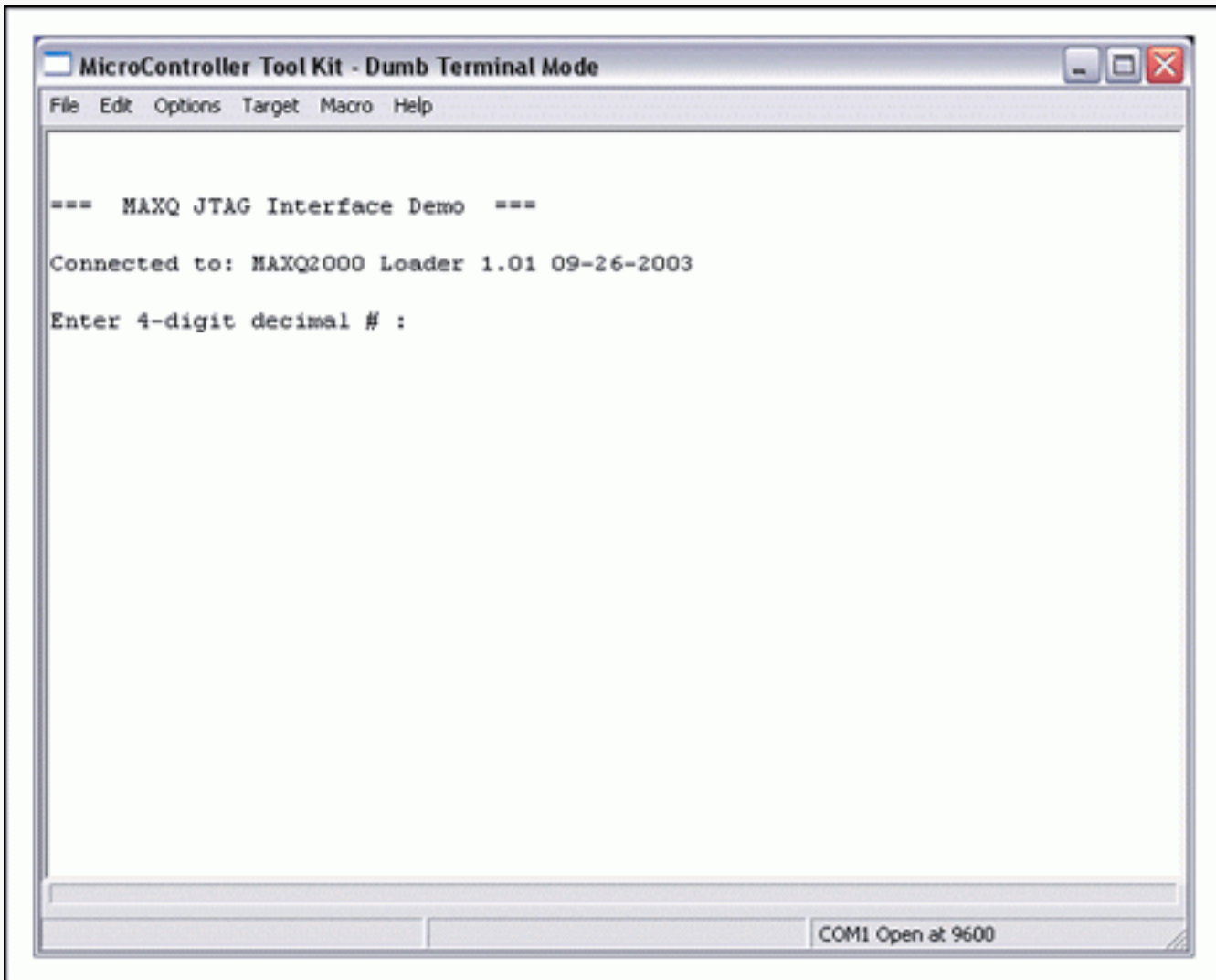


Figure 5. Demo application, serial-port prompt.

Now enter a 4-digit decimal number; there is no need to press ENTER afterwards. The demo application will complete its remaining operations (master erase, load code, and dump code) and will output the results, along with the hex values of the bytes dumped from program memory, as shown in **Figure 6**.

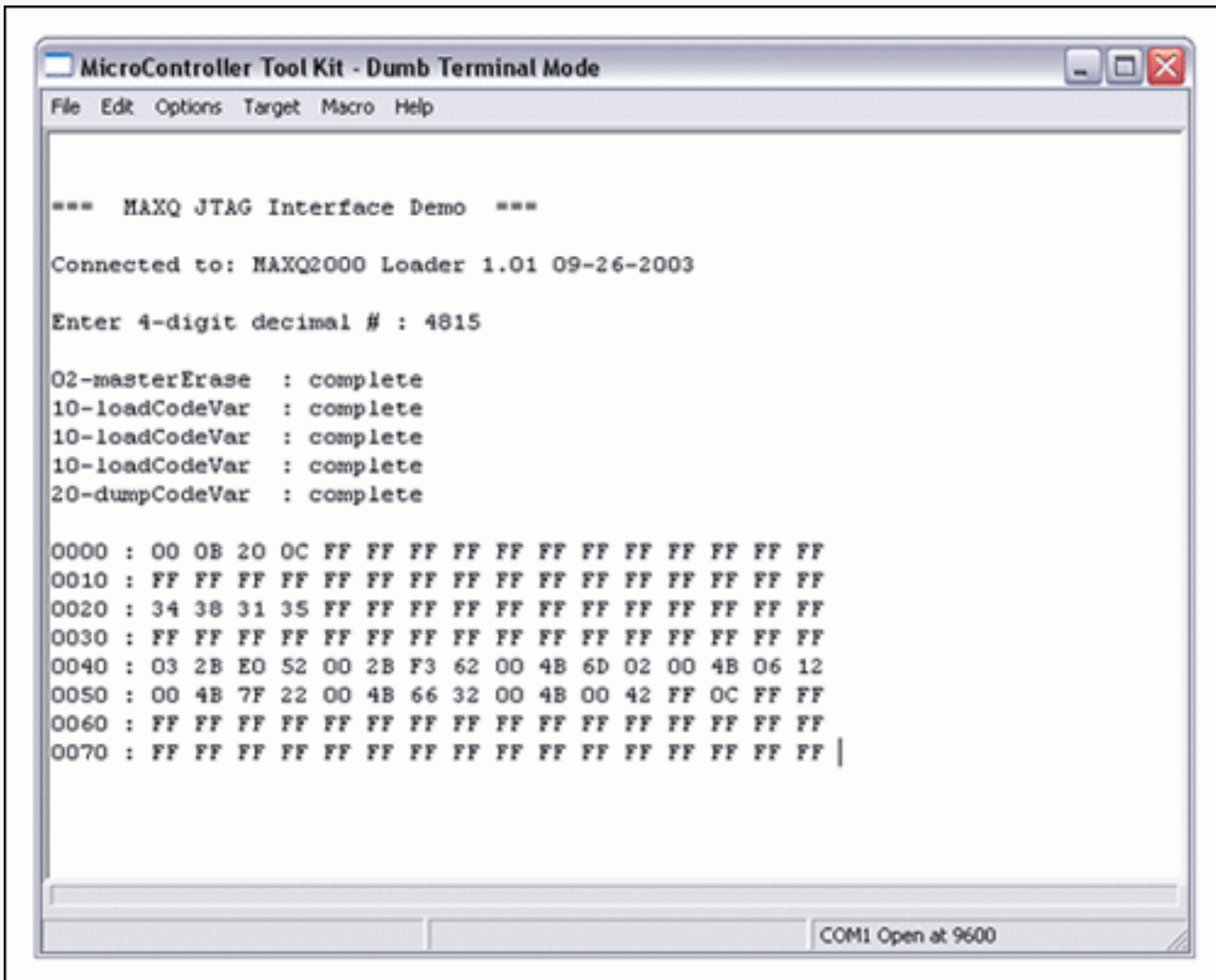


Figure 6. Demo application, serial-port output.

Conclusion

By using a set of standardized commands, the JTAG bootloader provided by MAXQ microcontrollers allows an external JTAG master to easily identify and program any MAXQ microcontroller. The code included with this application note can be used as a starting point to build a full-featured JTAG bootloader master application capable of identifying, initializing, loading, and verifying code and data memory contents of any MAXQ microcontroller which supports the standardized bootloader command set.

MAXQ is a registered trademark of Maxim Integrated Products, Inc.

SPI is a trademark of Motorola, Inc.

X-Acto is a registered trademark of Elmer's Products, Inc.

Related Parts

MAXQ2000: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

MAXQ3210: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

MAXQ3212: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

Automatic Updates

Would you like to be automatically notified when new application notes are published in your areas of interest?

[Sign up for EE-Mail™.](#)

Application note 4012: www.maxim-ic.com/an4012

More Information

For technical support: www.maxim-ic.com/support

For samples: www.maxim-ic.com/samples

Other questions and comments: www.maxim-ic.com/contact

AN4012, AN 4012, APP4012, Appnote4012, Appnote 4012

Copyright © by Maxim Integrated Products

Additional legal notices: www.maxim-ic.com/legal