Freescale Semiconductor
Application Note

AN1935
Rev. 1, 11/2005

Programming On-Chip Flash Memories of 56F80x Devices Using the JTAG/OnCE Interface

Reading and Writing Contents of Internal Flash Memory Units of 56F80x Devices Using the JTAG/OnCE Interface

Daniel Malik

1. Introduction

This Application Note describes the internal structure of the JTAG port and OnCE module and their functionality with respect to accessing the on-chip Flash memory units. The following sections describe algorithms which must be implemented and their implementation using C programming language.

2. JTAG Port and OnCE Module

2.1 General Description

The 56800 series of components provides board and chip-level testing capability through two on-chip modules, both accessed through the JTAG port/OnCE module interface:

- On-chip emulation (OnCE) module
- Test access port (TAP) and 16-state controller, also known as the JTAG port

Presence of the JTAG Port/OnCE module interface permits insertion of the chip into a target system while retaining debug control. This capability is especially important for devices without an external bus, because it eliminates the need for a costly cable to bring out the footprint of the chip required by a traditional emulator system.

Contents

The JTAG port is a dedicated user-accessible TAP, compatible with the *IEEE 1149.1a-1993 Standard Test Access Port and Boundary Scan Architecture*. Problems associated with testing high-density circuit boards have led to the development of this proposed standard under the sponsorship of the Test Technology Committee of IEEE and JTAG. The 56800 series of components supports circuit board test strategies based on this standard.

Five dedicated pins interface to the TAP, which contains a 16-state controller. The TAP uses a boundary scan technique to test the interconnections between integrated circuits after they are assembled onto a printed circuit board (PCB). Boundary scans allow a tester to observe and control signal levels at each component pin through a shift register placed next to each pin. This is important for testing continuity and determining if pins are stuck at the one or zero level.

Features of the TAP port include:

- Perform boundary scan operations to test circuit board electrical continuity
- Bypass the device for a given circuit board test by replacing the boundary scan register (BSR) with a single-bit register
- Sample the device system pins during operation and transparently shift out the result in the CSR; pre-load values to output pins prior to invoking the EXTEST instruction
- Disable the output drive to pins during circuit board testing
- Provide a means of accessing the OnCE module controller and circuits to control a target system
- Query identification information, manufacturer, part number, and version from a chip
- Force test data onto the outputs of a device IC while replacing its BSR in the serial data path with a single bit register
- Enable a weak pull-up current device on all input signals of a device IC, helping to assure deterministic test results in the presence of continuity fault during interconnect testing

The OnCE module is a Freescale-designed module used in Digital Signal Controller (DSC) chips to debug application software employed with the chip. The port is a separate on-chip block allowing non-intrusive device interaction with accessibility through the pins of the JTAG interface. The OnCE module makes it possible to examine registers, memory, or on-chip peripherals' contents in a special debug environment. This avoids sacrificing any user-accessible on-chip resources to perform debugging procedures. Additionally, on the 56F80x, the JTAG/OnCE port can be used to program the internal Flash memory OnCE module.

The capabilities of the OnCE module include the ability to:

- Interrupt or break into Debug Mode on a program memory address: fetch, read, write, or access
- Interrupt or break into Debug mode on a data memory address: read, write, or access
- Interrupt or break into Debug Mode on an on-chip peripheral register access: read, write, or access
- Enter Debug Mode using a device microprocessor instruction
- Display or modify the contents of any device core register
- Display or modify the contents of peripheral memory-mapped registers
- Display or modify any desired sections of program or data memory
- Trace one, single stepping, or as many as 256 instructions
- Save or restore the current state of the chip's pipeline
- Display the contents of the real-time instruction trace buffer, whether in Debug Mode or not
- Return to user mode from Debug Mode
- Set up breakpoints without being in Debug Mode
- Set hardware breakpoints, software breakpoints, and trace occurrences (OnCE events), possibly forcing the chip into Debug Mode; force a vectored interrupt; force the real-time instruction buffer to halt; or toggle a pin, based on the user's needs

2.2 JTAG/OnCE Pins

As described in the IEEE 1149.1a-1993 specification, the JTAG port requires a minimum of four pins to support TDI, TDO, TCK, and TMS signals. The 56F80x also uses the optional test reset ($\overline{\text{TRST}}$) input signal and a $\overline{\text{DE}}$ pin used for debug event monitoring. The pins and their functions are described in **Table 2-1**.

Table 2-1. Description of JTAG/OnCE Pins

Pin	Description
TDI	**Test Data Input** — This input provides a serial data stream to the JTAG and the OnCE module. It is sampled on the rising edge of TCK and has an on-chip pull-up resistor.
TDO	**Test Data Output** — This tri-stateable output provides a serial data stream from the JTAG and the OnCE module. It is driven in the Shift-IR and Shift-DR controller states of the JTAG state machine and changes on the falling edge of TCK.
TMS	**Test Mode Select Input** — This input sequences the TAP controller's state machine. It is sampled on the rising edge of TCK and has an on-chip pull-up resistor.
TCK	**Test Clock Input** — This input proves a gated clock to synchronize the test logic and shift serial data through the JTAG/OnCE port. The maximum frequency for TCK is 1/8 the maximum frequency of the 56F80x (i.e., 5MHz if the IP Bus clock is 40MHz). The TCK pin has an on-chip pull-down resistor.
$\overline{\text{TRST}}$	**Test Reset** — This input provides a reset signal to the TAP controller. This pin has an on-chip pull-up resistor.
$\overline{\text{DE}}$	**Debug Event** — Assertion of this output signals that the OnCE event has occurred

2.3 JTAG Port Architecture, Timing of Signals and State Machine

The TAP controller is a simple 16-state machine used to sequence the JTAG port through its valid operations:

- Serially shift JTAG port instructions in or out and decode them
- Serially input or output a data value
- Update a JTAG port (or OnCE module) register

The block diagram of the JTAG port is shown in **Figure 2-1**. The JTAG port has four read/write registers: the Instruction Register, the Boundary Scan Register, the Device Identification Register, and the Bypass Register. The JTAG port also provides a path for accessing the OnCE module.

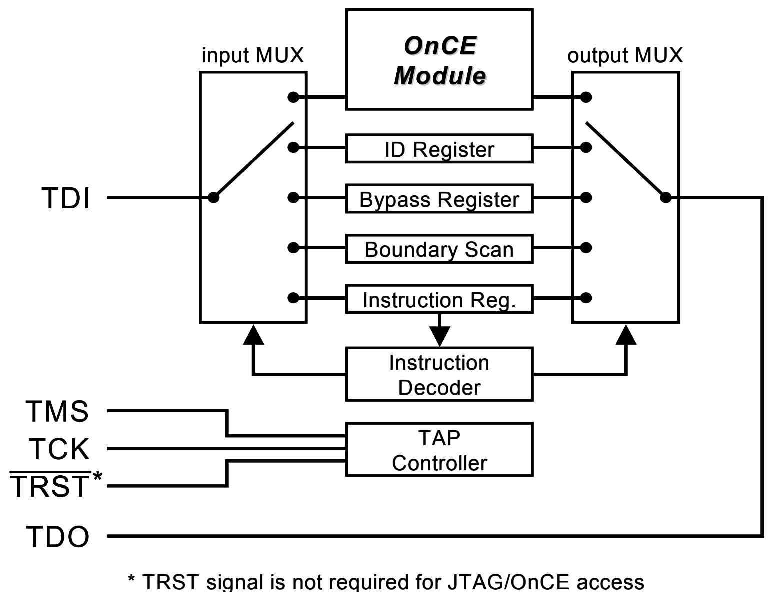

Figure 2-1. Block Diagram of the JTAG Port

Timing of the JTAG signals is shown in **Figure 2-2**. The TDO pin remains in the high impedance state except during the shift-DR or shift-IR controller states. In these controller states, TDO is updated on the falling edge of TCK. TDI and TMS are sampled on the rising edge of TCK.

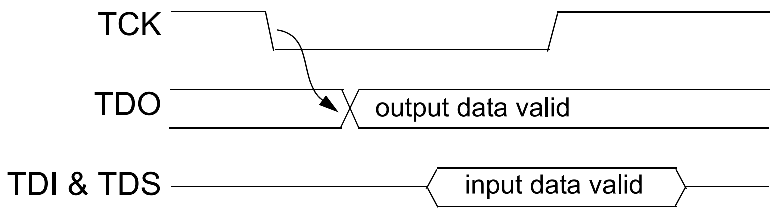

Figure 2-2. Timing of Signals on JTAG Port Pins

The TAP controller is a synchronous finite-state machine containing sixteen states, as illustrated in **Figure 2-3**. The TAP controller responds to changes at the TMS and TCK signals. Transitions from one state to another occur on the rising edge of TCK. The value shown adjacent to each state transition in this figure represents the signal present at TMS at the time of a rising edge at TCK.

There are two paths through the 16-state TAP machine. The Instruction path captures and loads JTAG instructions into the Instruction Register. The Data path captures and loads data into the other JTAG registers and also provides a path for communicating with the OnCE module. The TAP controller executes the last instruction decoded until a new instruction is entered at the Update-IR state, or until the Test-Logic-Reset state is entered. When using the JTAG port to access OnCE module registers, accesses are first enabled by shifting the ENABLE_ONCE instruction into the JTAGIR. After this is selected, the OnCE module registers and commands are read and written through the JTAG pins using the Data path. Asserting the JTAG's $\overline{\text{TRST}}$ pin asynchronously forces the JTAG state machine into the Test-Logic-Reset state.

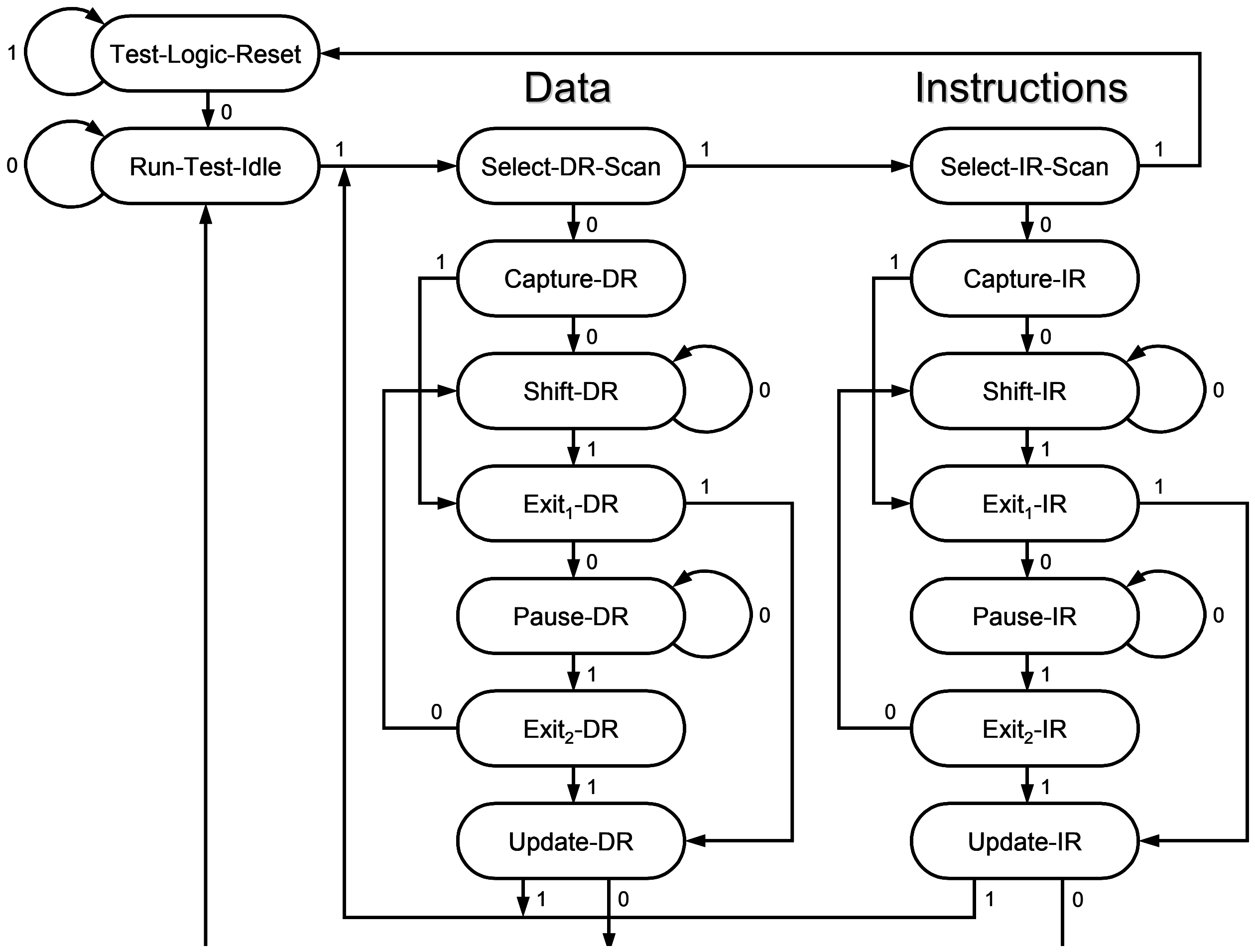

Figure 2-3. Description of the TAP State Machine

3. Algorithms for Accessing the JTAG Port

Algorithms in the following sections represent one of the possible approaches for accessing the Flash memory units. They do not explore the full capabilities of the JTAG port and the OnCE module and were created solely for the purpose of programming the on-chip Flash memories of the target processor. Very little speed optimization was done for educational purposes and these algorithms may prove to be too slow for a high-volume production environment.

These algorithms are not platform-specific. They can be used to create an application running on a PC, a tester machine, a microprocesor system or any other platform which is convenient to use. Almost any platform with enough I/O pins connecting it to the JTAG environment can be turned into a device Flash programmer.

An application for the PC Windows enviroment was created based on an optimized version of the algorithms to achieve minimum programming time and to support situations where the device shares the JTAG chain with other devices and other features. For additional information, see References, item [2].

3.1 Primitives for Accessing the JTAG Pins

All algorithms presented here rely on primitives which access the JTAG port pins. These macros or functions are platform specific and their implementation is up to the user. Signals TDI, TMS and TCK are considered outputs and signal TDO is considered input. Use of the $\overline{\text{TRST}}$ and $\overline{\text{DE}}$ signals is not required for accessing the

JTAG/OnCE and these signals are not used in the algorithms. The user is expected to reset the JTAG TAP state machine to Test-Logic-Reset state at power-up by asserting the $\overline{\text{TRST}}$ pin as indicated in the chip datasheet. Where hardware measures are provided on the proprietary target platform for asserting the $\overline{\text{TRST}}$ pin, external connection to this pin is not required.

The primitives are:

- JTAG_TCK_SET
- JTAG_TCK_RESET
- JTAG_TMS_SET
- JTAG_TMS_RESET
- JTAG_TDI_SET
- JTAG_TDI_RESET
- JTAG_TDO_VALUE

The JTAG_XXX_SET primitives assert the respective signal (logical Hi). The JTAG_XXX_RESET primitives deassert the respective signal (logical Lo). The JTAG_TDO_VALUE primitive returns a value of 0 or 1 when the TDO pin is in logical Hi or Lo state, respectively.

Based on TDI-related primitives, it’s possible to define one more:

```
#define JTAG_TDI_ASSIGN(i)if (i&0x0001) JTAG_TDI_SET; else JTAG_TDI_RESET
```

This primitive asserts the TDI signal for all odd arguments and deasserts it for all even arguments.

3.2 Executing JTAG Instructions

The JTAG port contains a 4-bit wide Instruction Register. Instructions are transferred into this register during the Shift-IR state of the TAP state machine and are decoded by entering the Update-IR state of the TAP. The JTAG controller executes the last decoded instruction until a new one is entered and decoded. The instructions as well as data are entered serially through the TDI pin, LSB first.

The JTAG instructions and their binary codes are shown in **Table 3-1**. Only a subset of these JTAG instructions will be required for programming the on-chip Flash memories as described later in this Application Note.

Table 3-1. JTAG Instructions

Code (binary)	Instruction
0000	EXTEST
0001	SAMPLE/PRELOAD
0010	IDCODE
0011	EXTEST_PULLUP
0100	HIGHZ
0101	CLAMP
0110	ENABLE_ONCE
0111	DEBUG_REQUEST

Table 3-1. JTAG Instructions (Continued)

Code (binary)	Instruction
1111	BYPASS

While a new instruction is shifted in through the TDI pin, the TDO pin outputs status information. The status has the following 4-bit format:

OS1	OS0	0	1

The LSB is shifted out first. The OS0 and OS1 bits indicate the current state of the device; see Table 3-2.

Table 3-2. JTAG Status

OS1	OS0	Description
0	0	Normal operation: device core executing instructions or in reset
0	1	Stop/Wait: device core in Stop or Wait Mode
1	0	Busy: device is performing external or peripheral access (wait states)
1	1	Debug: device core halted and in Debug Mode

IDCODE Instruction

The IDCODE instruction enables the 32-bit wide ID Register between TDI and TDO. It is provided as a public instruction that allows the determination of the manufacturer, part number, and version of a component through the TAP.

The instruction is not really necessary for accessing the Flash memories, but is useful for determining the part number and version of the attached chip.

DEBUG_REQUEST Instruction

The DEBUG_REQUEST instruction asserts a request to halt the core for entry to Debug Mode. It is typically used in conjunction with ENABLE_ONCE to perform system debug functions. It is provided as a public instruction. When the DEBUG_REQUEST instruction is invoked, the TDI and TDO pins are connected to the bypass register.

ENABLE_ONCE Instruction

The ENABLE_ONCE instruction enables the JTAG port to communicate with the OnCE state machine and registers. It is provided to allow the user to perform system debug functions. When the ENABLE_ONCE instruction is invoked, the TDI and TDO pins are connected directly to the OnCE registers. The particular OnCE register connected between TDI and TDO is selected by the OnCE state machine and the OnCE instruction being executed. All communication with the OnCE instruction controller is done through the Data path of the JTAG state machine.

To execute the JTAG instruction, bring the TAP state machine to the Shift-IR phase, shift in the new instruction and bring the TAP state machine to the Update-IR state to decode the new instruction. Implementation of this algorithm is demonstrated in Code Example 3-1.

Code Example 3-1. Execution of JTAG Instruction

```
/* Execution of Jtag instruction */
/* expects Test-Logic-Reset or Run-Test-Idle state on entry */
/* leaves the TAP in Run-Test-Idle on exit */
/* returns Jtag status */
int jtag_instruction_exec(int instruction) {
     int i,status=0;
     JTAG_TCK_SET;
     JTAG_TMS_RESET;                  /* Go to Run-Test-Idle */
     JTAG_TCK_RESET;
     JTAG_TCK_SET;
     JTAG_TMS_SET;                    /* Go to Select-DR-Scan */
     JTAG_TCK_RESET;
     JTAG_TCK_SET;
     JTAG_TCK_RESET;
     JTAG_TCK_SET;                    /* Go to Select-IR-Scan */
     JTAG_TMS_RESET;                  /* Go to Capture-IR */
     JTAG_TCK_RESET;
     JTAG_TCK_SET;
     JTAG_TCK_RESET;
     JTAG_TCK_SET;                    /* Go to Shift-IR */
     JTAG_TCK_RESET;                  /* TAP is now in Shift-IR state */
     for (i=0;i<4;i++) {
          JTAG_TDI_ASSIGN(instruction);
          instruction>>=1;
          if (i==3) JTAG_TMS_SET;     /* Go to Exit1-IR */
          JTAG_TCK_SET;
          status>>=1;
          status|=JTAG_TDO_VALUE<<3;
          JTAG_TCK_RESET;
     }
     JTAG_TCK_SET;                    /* Go to Update-IR */
     JTAG_TMS_RESET;                  /* Go to Run-Test-Idle */
     JTAG_TCK_RESET;
     JTAG_TCK_SET;
     return(status);
}
```

3.3 Transferring Data To and From the JTAG Port

After storing a JTAG instruction in the IR register and executing it, it's usually necessary to transfer data associated with the instruction. Data is shifted in and out of the selected JTAG register or OnCE module in the Shift-DR state of the TAP state machine. The data is then captured in the selected register by entering the Update-DR state of the TAP. The length of the data register depends on the JTAG instruction being executed. The function in **Code Example 3-2** enables transfer of variable length data.

Code Example 3-2. Transfer of Data In and Out of the JTAG Data Registers

```
/* Shifts up to 32 bits in and out of the jtag DR path */
/* expects Test-Logic-Reset or Run-Test-Idle state on entry */
/* and leaves the TAP in Run-Test-Idle on exit */
unsigned long jtag_data_shift(unsigned long data, int bit_count) {
     int i; unsigned long result=0;
     JTAG_TCK_SET;
     JTAG_TMS_RESET;                  /* Go to Run-Test-Idle */
     JTAG_TCK_RESET;
     JTAG_TCK_SET;
     JTAG_TMS_SET;                    /* Go to Select-DR-Scan */
```

```
        JTAG_TCK_RESET;
        JTAG_TCK_SET;
        JTAG_TMS_RESET;                /* Go to Capture-DR */
        JTAG_TCK_RESET;
        JTAG_TCK_SET;
        JTAG_TCK_RESET;
        JTAG_TCK_SET;                  /* Go to Shift-DR */
        JTAG_TCK_RESET;                /* TAP is now in Shift-DR state */
        for (i=0;i<bit_count;i++) {
             JTAG_TDI_ASSIGN(data);
             data>>=1;
             if (i==(bit_count-1)) JTAG_TMS_SET;    /* Go to Exit1-DR */
             JTAG_TCK_SET;
             result>>=1;
             result|=((unsigned long int)JTAG_TDO_VALUE)<<(bit_count-1);
             JTAG_TCK_RESET;
        }
        JTAG_TCK_SET;                  /* Go to Update-DR */
        JTAG_TMS_RESET;                /* Go to Run-Test-Idle */
        JTAG_TCK_RESET;
        JTAG_TCK_SET;
        return(result);
   }
```

3.4 Preparing for OnCE Module Access

The algorithms needed to operate the JTAG port have now been created. The function in **Code Example 3-3** reads the JTAG ID of the target device, issues the DEBUG_REQUEST and ENABLE_ONCE commands and waits until the core enters the Debug Mode. After execution of the ENABLE_ONCE command, communication with the OnCE module can begin.

Code Example 3-3. Preparation for OnCE Access

```
 /* Brings target into the Debug mode and enables the OnCE interface */
 void init_target (void) {
      int status,i;
      unsigned long int result;
      status=jtag_instruction_exec(0x2);        /* IDCODE */
      printf("IDCode status: %#x\r\n",status);
      result=jtag_data_shift(0,32);
      printf("Jtag ID: %#lx\r\n",result);
      status=jtag_instruction_exec(0x7);        /* Debug Request */
      printf("Debug Request status: %#x\r\n",status);
      while (jtag_instruction_exec(0x6)!=0xd);  /* Enable OnCE, wait */
 }
```

4. Algorithms for Communication with the OnCE Module

While the JTAG port provides board test capability, the OnCE module provides emulation and debug capabilities. The OnCE module permits full-speed, non-intrusive emulation on a target system.

The JTAG and OnCE blocks are tightly coupled. The JTAG port is the master and must enable the OnCE module before the OnCE module can be accessed.

The OnCE module has its own instruction register (OCMDR) and instruction decoder. After a command is latched into the OCMDR, the command decoder implements the instruction through the OnCE state machine and control block. There are two types of commands:

1. Read commands, causing the chip to deliver required data
2. Write commands, transferring data into the chip, then writing it in one of the on-chip resources

The commands are eight bits long and have the format displayed in **Table 4-1**. The lowest five bits, RS0 - RS4, identify the source for the operation, described in **Table 4-2**. Bits 5, 6, and 7 contain the exit bit, EX, the execute bit, GO, and the read/write bit, R/$\overline{\text{W}}$.

Table 4-1. OnCE Command Format

7	6	5	4	3	2	1	0
R/$\overline{\text{W}}$	GO	EX	RS4	RS3	RS2	RS1	RS0

Table 4-2. OnCE Register Selection Encoding

RS4 - RS0	Register or Action Selected	Available in Mode	Type of Access Allowed
00000	No register selected	All	N/A
00001	OnCE Breakpoint and Trace Counter (OCNTR)	All	Read/Write
00010	OnCE Debug Control Register (OCR)	All	Read/Write
00100	OnCE Breakpoint Address Register (OBAR)	All	Write
01000	OnCE PGDB Bus Transfer Register (OPGDBR)	Debug	Read
01001	OnCE Program Data Bus Register (OPDBR)	Debug	Read/Write
01010	OnCE Program Address Register—Fetch cycle (OPABFR)	FIFO halted	Read
01100	Clear OCNTR	All	N/A
10000	OnCE Program Address Register—Execute cycle (OPABER)	FIFO halted	Read
10001	OnCE Program address FIFO (OPFIFO)	FIFO halted	Read
10011	OnCE Program Address Register—Decode cycle (OPABDR)	FIFO halted	Read

When the exit bit, EX, is set, the device core will exit the Debug processing state after the command is executed; otherwise, the Debug state is preserved. The execute bit, GO, signals that the device core instruction should be executed. The read/write bit, R/$\overline{\text{W}}$, indicates whether a read or write operation should be performed with the register selected by the RS bits.

It is possible to define a new macro for executing OnCE commands:

```
#define once_instruction_exec(instruction, rw, go, ex)
        jtag_data_shift(instruction|(ex<<5)|(go<<6)|(rw<<7), 8)
```

Once the command is transferred into the OnCE module, it's necessary to read or write contents of the selected register. As with the JTAG instructions, only a subset of OnCE commands is needed when programming the on-chip Flash memories. In fact, only two OnCE commands will be used: *Write to Program Data Bus Register (OPDBR) and Read from OnCE PGDB Bus Transfer Register (OPGDBR)*. The first command executes individual instructions on the device core and the second command transfers data out of the device core. As only access to a 16-bit register is required, use very simple macros:

```
#define once_data_write(data)jtag_data_shift(data,16)
#define once_data_read()jtag_data_shift(0,16)
```

4.1 Executing One-Word Instructions

To force execution of a one-word instruction from the Debug Mode, write the *OPDBR* with the opcode of the instruction to be executed and set GO = 1 and EX = 0. The instruction is then executed. During instruction execution, the OS status bits in the JTAG status equal 00. Upon completion, OS1:OS0 = 11, the Debug Mode. Typically, the period of time OS1:OS0 = 00 is unnoticeably small.

To define a new macro for executing one-word instructions:

```
#define once_execute_instruction1(opcode)
          once_instruction_exec(0x09,0,1,0); once_data_write(opcode)
```

4.2 Execution Two-Word Instructions

To force execution of a two-word instruction from the Debug Mode, write the *OPDBR* with the opcode of the instruction to be executed and set GO = EX = 0. Next, write *OPDBR* with the operand with GO = 1 and EX = 0; the instruction then executes. As in the one-word case, JTAG status can be polled to examine the execution.

```
#define once_execute_instruction2(opcode, operand)
                                       once_instruction_exec(0x09,0,0,0);
                                       once_data_write(opcode);
                                       once_instruction_exec(0x09,0,1,0);
                                       once_data_write(operand)
```

4.3 Instruction Set Supported by the OnCE Module

The set of supported instructions for execution from the Debug Mode, GO, but not EX, is:

- JMP #xxxx
- MOVE #xxxx,register
- MOVE register,x:0xFFFF
- MOVE register,register
- MOVE register,x:(Rx)+
- MOVE x:(Rx)+,register
- MOVE register,p:(Rx)+
- MOVE p:(Rx)+,register

Execution of other device instructions is possible, but only the preceding set are specified and supported. Three-word instructions cannot be executed from Debug Mode.

4.4 Reading Data Out of the Device Core

As indicated in the set of supported instructions, it is possible from Debug Mode to write into the *OPGDB Register* located at address x:0xFFFF. Contents of this register can be then transferred out of the device using the *Read from OnCE PGDB Bus Transfer Register* command.

```
#define once_opgdbr_read()
                        (once_instruction_exec(0x08,1,1,0), once_data_read())
```

4.5 Instruction Execution - Examples

Because there is such a high number of possible register combinations, only a subset of all instructions supported in Debug Mode are listed here:

```
/* NOP */
#define once_nop()                     once_execute_instruction1(0xe040)
/* MOVE <data>,Y0 */
#define once_move_data_to_y0(data)     once_execute_instruction2(0x87c1,data)
/* MOVE <data>,R0 */
#define once_move_data_to_r0(data)     once_execute_instruction2(0x87d0,data)
/* MOVE Y0,x:address */        /* NOTE: only address 0xFFFF is supported */
#define once_move_y0_to_xmem(address)
                               once_execute_instruction2(0xd154,address)
/* MOVE x:(R0)+,Y0 */
#define once_move_xr0_inc_to_y0()      once_execute_instruction1(0xf100)
/* MOVE Y0,x:(R0)+ */
#define once_move_y0_to_xr0_inc()      once_execute_instruction1(0xd100)
/* MOVE R0,Y0 */
#define once_move_r0_to_y0()           once_execute_instruction1(0x8110)
/* MOVE OMR,Y0 */
#define once_move_omr_to_y0()          once_execute_instruction1(0x8118)
/* MOVE Y0,OMR */
#define once_move_y0_to_omr()          once_execute_instruction1(0x8881)
/* MOVE Y0,p:(R0)+ */
#define once_move_y0_to_pr0_inc()      once_execute_instruction1(0xe100)
/* MOVE p:(R0)+,Y0 */
#define once_move_pr0_inc_to_y0()      once_execute_instruction1(0xe120)
```

5. Algorithms for Accessing the Flash Memory

The Flash memory blocks present on the 56F80x devices are erased and programmed using dedicated Flash Interface Units (FIU). Each of the Flash memories has its own FIU; placement of the respective FIUs in the device's memory map can be found in References, item [1]. The algorithms presented in this section use "intelligent", rather than "dumb", erase and programing; see References, item [1] for details.

The FIUs are accessed by executing instructions on the device core in Debug Mode.

5.1 Timing of Flash Program/Erase

Timing of the Flash program and erase cycles is governed by a set of timing registers which are part of the FIU. The timebase for all the timings is created by the IPBus Clock of the chip, which is dependent on the On-Chip Clock Synthesis (OCCS) block set-up. After chip Reset or power-up, the IPBus Clock receives half of the frequency present on the XTAL pin of the chip. In the usual set-up, the chip is provided with an 8MHz crystal and therefore the IPBus Clock equals 4MHz after power-up or Reset.

The reset values of the FIU timing registers are optimized for full-speed operation of the chip when the IP Bus Clock receives 40MHz. To prevent overstress and possible permanent damage of the Flash memories, either the OCCS unit must be reprogrammed to supply 40MHz to the IP Bus Clock, or the timing registers need to be reprogrammed with new values suitable for the lower IP Bus Clock frequencies.

The algorithm for initialization of the FIU timing registers is shown in **Code Example 5-1**.

Code Example 5-1. Initialization of FIU Timing Registers

```
/* initialises the FIU Timing registers */
void once_init_flash_iface(unsigned int fiu_address) {
     unsigned int i;
     printf("Initialising FIU at address: %#x\r\n",address);
     once_move_data_to_r2(address);      /* MOVE #<base address>,R2 */
     once_move_data_to_y0(0);            /* MOVE #0,Y0              */
     once_move_y0_to_xr2_inc();          /* clear FIU_CNTL register */
     once_move_y0_to_xr2_inc();          /* clear FIU_PE register   */
     once_move_y0_to_xr2_inc();          /* clear FIU_EE register   */
     once_move_data_to_r0(fiu_address+8);/* MOVE #<fiu_address+8>,R0*/
     once_move_data_to_y0(FIU_CLKDIVISOR);/* fill timing regs       */
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TERASEL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TMEL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TNVSL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TPGSL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TPROGL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TNVHL);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TNVHL1);
     once_move_y0_to_xr0_inc();
     once_move_data_to_y0(FIU_TRCVL);
     once_move_y0_to_xr0_inc();
     printf("FIU (%#x) initialisation done.\r\n", fiu_address);
}
```

Values for the timing registers at 40MHz and 4MHz of IPBus Clock frequencies are shown in **Table 5-1**.

Table 5-1. Values of FIU Timing Registers

Register	Reset Values (40MHz)	Values for 4MHz	Time Corresponding to the Reset Value
FIU_CLKDIVISOR	15	15	N/A
FIU_TERASEL	15	2	26.2ms
FIU_TMEL	31	6	52.4ms
FIU_TNVSL	255	26	6.4µs
FIU_TPGSL	511	51	12.8µs

Table 5-1. Values of FIU Timing Registers

Register	Reset Values (40MHz)	Values for 4MHz	Time Corresponding to the Reset Value
FIU_TPROGL	1023	102	25.6µs
FIU_TNVHL	255	26	6.4µs
FIU_TNVHL1	4095	410	102.4µs
FIU_TRCVL	63	6	1.6µs

5.2 Mass Erasing the Flash Memory

The unprogrammed (erased) state of any Flash memory bit is 1. Individual bits can be programmed to the 0 state at any time; however, in order to return even a single bit to the erased state, the whole memory page containing the bit and consisting of 256 memory words must be erased. Instead of erasing only one memory page, the FIU offers the possibility of erasing the whole memory in a single erase operation, called mass erase.

To perform the Flash mass erase operation, follow these steps:

- Enable erasing by setting the IEE bit and set the page number in the FIU_EE register to 0 **Exception:** when mass erasing bootflash of the 56F807, set the page to 0x78.
- Set the MAS1 bit in the FIU_CNTL register
- While the IEE bit is set, write any value to an address into the page 0 (0x78). This write to the Flash memory map will start the FIU internal state machine, running the Flash through its erase process
- Do not attempt to access Flash again until the BUSY signal clears in the FIU_CNTL register
- Ensure that the FIU_CNTL and FIU_EE registers are cleared when finished

The algorithm for performing the mass erase operation is shown in Code Example 5-2.

Code Example 5-2. Flash Memory Mass Erase

```
/* Performs mass erase */
void once_flash_mass_erase(unsigned int fiu_address, unsigned int addr){
    once_move_data_to_r0(addr);          /* MOVE #<address>,R0        */
    once_move_data_to_r1(fiu_address+2);/* MOVE #<base address+2>,R1 */
    #ifdef DSP56F807
    if (fiu_address==BFIU)               /* 807 BFIU: see [1.] p.5-15 */
        {once_move_data_to_y0(0x4078);} /* MOVE #<ee>,Y0             */
    else
        {once_move_data_to_y0(0x4000);} /* MOVE #<ee>,Y0             */
    #else
    once_move_data_to_y0(0x4000);        /* MOVE #<ee>,Y0             */
    #endif
    once_move_y0_to_xr1_inc();           /* MOVE Y0,x:R1 (FIU_EE)     */
    once_move_data_to_r1(fiu_address);   /* MOVE #<base address>,R1   */
    once_move_data_to_y0(0x0002);        /* MOVE #<cntl>,Y0           */
    once_move_y0_to_xr1_inc();           /* MOVE Y0,x:R1 (FIU_CNTL)   */
    if (fiu_address==DFIU)
        {once_move_y0_to_xr0_inc();}     /* MOVE Y0,x:R0 (wr x:addr)  */
    else
        {once_move_y0_to_pr0_inc();}     /* MOVE Y0,x:R0 (wr p:addr)  */
    do {
        once_move_data_to_r1(fiu_address);/* MOVE #<fiu_address>,R1 */
        once_nop();                      /* NOP                       */
        once_move_xr1_inc_to_y0();      /* MOVE x:R1,Y0              */
        once_move_y0_to_xmem(0xffff);   /* MOVE Y0,<OPGDBR>          */
    } while (once_opgdbr_read()&0x8000);/* repeat while BUSY is set  */
    once_move_data_to_r1(fiu_address+2);/* MOVE #<base address+2>,R1 */
    once_move_data_to_r0(fiu_address);  /* MOVE #<base address>,R0   */
    once_move_data_to_y0(0);            /* MOVE #0,Y0                */
    once_move_y0_to_xr0_inc();          /* MOVE Y0,x:R0 (FIU_CNTL)   */
    once_move_y0_to_xr1_inc();          /* MOVE Y0,x:R1 (FIU_EE)     */
    printf("Flash (%#x) mass erase done.\r\n", fiu_address);
}
```

5.3 Programming the Flash Memory

The “intelligent” programming algorithm allows for only one word at a time to be programmed into the Flash memory. The “dumb” algorithm allows for up to 32 words to be programmed at once and is therefore faster. However, this mode is sensitive to the exact timing of all the operations and the Flash unit can easily be overstressed.

To perform the intelligent one-word programming operation, follow these steps:

- Enable programming by setting the IPE bit and row number in the FIU_PE register. To calculate the row number, use the following algorithm:
  - Target_Address AND 0x7FFF divided by 0x20 equals ROW
  - Or, put differently, set the MSB of the target address to zero and right shift the result five bits
- Write the value desired to the proper word in the Flash memory map. A single location in the Flash may map to different locations in the memory map based upon the mode selected on startup; the FIU will adjust accordingly. While the IPE bit is set, this write to the Flash memory map starts the internal state machine to run the Flash through its programming process.
- Do not attempt to access the Flash again until the BUSY signal clears in the FIU_CNTL register
- When programming words has been completed, remember to clear the IPE bit in the FIU_PE register

An algorithm for performing the one word programming operation and verification is shown in **Code Example 5-3**.

Code Example 5-3. Flash Memory Programming

```
/* Programs and verifies one word of internal flash memory */
int once_flash_program_1word(unsigned int fiu_address, unsigned int addr, unsigned
int data) {
    unsigned int i;
    once_move_data_to_r1(fiu_address+1);/* MOVE #<fiu_address+1>,R1  */
    once_move_data_to_r0(addr);         /* MOVE #<address>,R0        */
    once_move_data_to_y0(0x4000 + (( addr >> 5) & 0x03ff));
                                        /* MOVE #<pe>,Y0             */
    once_move_y0_to_xr1_inc();          /* MOVE Y0,x:R1 (FIU_PE)     */
    once_move_data_to_y0(data);         /* MOVE #<data>,Y0           */
    if (fiu_address==DFIU)
        {once_move_y0_to_xr0_inc();}    /* MOVE Y0,x:R0 (x:addr)     */
    else
        {once_move_y0_to_pr0_inc();}    /* MOVE Y0,x:R0 (p:addr)     */
    once_move_data_to_r0(addr);         /* MOVE #<address>,R0        */
    do {
        once_move_data_to_r1(fiu_address);/* MOVE #<fiu_address>,R1 */
        once_nop();                     /* NOP                       */
        once_move_xr1_inc_to_y0();      /* MOVE x:R1,Y0              */
        once_move_y0_to_xmem(0xffff);   /* MOVE Y0,<OPGDBR>            */
    } while (once_opgdbr_read()&0x8000);/* repeat while BUSY is set  */
    once_move_data_to_r1(fiu_address+1);/* MOVE #<base address+1>,R1 */
    once_move_data_to_y0(0);            /* MOVE #0,Y0                */
    once_move_y0_to_xr1_inc();          /* MOVE Y0,x:R1 (FIU_PE)     */
    if (fiu_address==DFIU)
        {once_move_xr0_inc_to_y0();}    /* MOVE x:R0,Y0 (x:addr)     */
    else
        {once_move_pr0_inc_to_y0();}    /* MOVE Y0,x:R0 (p:addr)     */
    once_move_y0_to_xmem(0xffff);       /* MOVE Y0,<OPGDBR>          */
    if ((i=once_opgdbr_read())!=data) { /* Read OPGDBR register      */
        printf("Pgm error @ %#x, wr: %#x, rd: %#x\r\n", addr, data, i);
        return(1);
    }
    return(0);
}
```

6. Conclusion

In **Sections 3.**, **4.**, and **5.**, a whole library of functions and macros was built, which enables erasing, programming and verifying contents of the internal Flash memories over the JTAG/OnCE interface. Since the JTAG signals can be as fast as 5MHz, programming time below 5 seconds can be achieved with the 56F805. Therefore, the programming technique described here is suitable for even a high-volume production environment.

7. References

[1.] 56F80x 16-bit Digital Signal Processor, User's Manual, DSP56F801-7UM, Rev. 3.0, Freescale Semiconductor, Inc.

[2.] 56F800 Flash Programming via JTAG/OnCE using the Parallel Command Converter, Rev. 0.4, Freescale Semiconductor, Inc.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN1935
Rev. 1
11/2005