
Using an AT91EB40A Evaluation Board to Control an AT91 ARM7TDMI[®] Processor Via the JTAG-ICE Interface

Introduction

This application note describes how to use an AT91EB40A Evaluation Board based on the AT91R40008 microcontroller to control an AT91 ARM7TDMI processor via the standard ARM JTAG-ICE Interface. The objective is to elicit the microcontroller status, to read or write processor registers, to read or write a word into internal or external memory, and obtain the status of the internal peripherals. These functions provide control of the AT91 ARM7TDMI processor.

A brief description of using JTAG to reload an AT91 evaluation board is provided in the section "Application Example: Flash Up-loader" on page 18.

The AT91 ARM7TDMI processor can be driven via a JTAG-style serial interface to serially insert instructions into the processor's pipeline without using an external data bus.

The AT91 ARM7TDMI JTAG interface is based on IEEE Std. 1149.1- 1990, "Standard Test Access Port and Boundary-Scan Architecture". Please refer to this standard for an explanation of the terms used in this application note and for a description of the Test Access Port (TAP) controller states.

References

Information included in this Application Note is derived from the following sources:

1. IEEE Std. 1149.1-1990 "Standard Test Access Port and Boundary-Scan Architecture"
2. ARM Architecture Reference Manual; 2nd edition, Addison Wesley Professional Copyright, 2001
3. ATMEL ARM7TDMI Datasheet; Revision 0673B, January 1998
4. ATMEL AT91EB40A Evaluation Board User Guide; Revision 2635B, 24 June, 2002
5. ARM Ltd. application note EAN-37 "Embedded ICE Debugging of Low-speed ARM7TDMI Systems".



**AT91 ARM[®]
Thumb[®]
Microcontrollers**

**Application
Note**

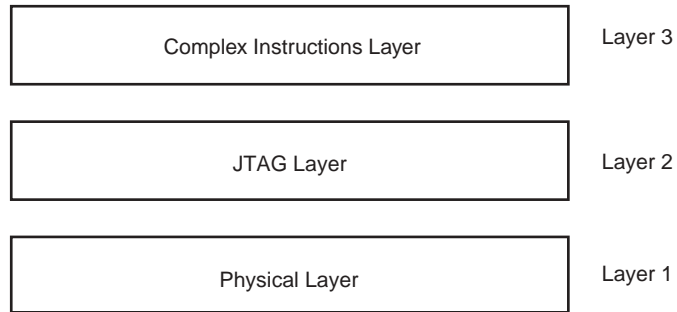
Rev. 2668A-ATARM-14-Jan-03



Three-layer Model of AT91 ARM7TDMI

Driving an AT91 ARM7TDMI processor is represented by the layer model in Figure 1.

Figure 1. Driving an AT91 ARM7TDMI Processor Model



Physical Layer

The physical layer is linked to the JTAG-style serial interface. The JTAG protocol requires four JTAG signals: TDO, TDI, TCK and TMS to drive an ARM7TDMI. These signals are generated on four PIOs of an AT91 EB40A Evaluation Board based on the AT91R40008 microcontroller. Please refer to the document AT91EB40A Evaluation Board User Guide; Revision 2635B, 24 June, 2002, Chapter "JTAG Interface" and the Appendix Schematics "Reset and JTAG Interface".

Table 1 below represents the HE-10 JTAG connector of the AT91 evaluation board target and gives a description of the various signals for ARM targets using a 20-pin debug connection.

Table 1. HE-10 JTAG Connector: $V_{DD} = 3.3V$, $GND = 0V$, $NC = \text{Not Connected}$

20-pin Debug Connection									
1	3	5	7	9	11	13	15	17	19
V_{DD}	NC	TDI	TMS	TCK	TCK	TDO	V_{DD}	NC	NC
V_{DD}	GND	GND	GND	GND	GND	GND	GND	GND	GND
2	4	6	8	10	12	14	16	18	20

The PIOs must be connected to pins 5, 7, 9 and 13 of the connector. The voltage on the PIOs is 3.3V.

Pin 11 is connected to pin 9 but the TCK signal must be placed on pin 9.

A software reset of the JTAG can be obtained by using TMS and TCK, therefore the JTAG reset signal is not necessary. The TRST signal should be connected to pin 15.

JTAG Layer

Scan Chains and JTAG Interface

There are three JTAG-style scan chains inside the ARM7TDMI which provide:

- Testing
- Debugging
- ICEBreaker programming

The scan chains are controlled from a JTAG-style Test Access Port (TAP) controller. For further details of the JTAG specification, please refer to IEEE Std. 1149.1- 1990

"Standard Test Access Port and Boundary-Scan Architecture" and to the "Debug Interface" chapter of the Atmel ARM7TDMI Datasheet.

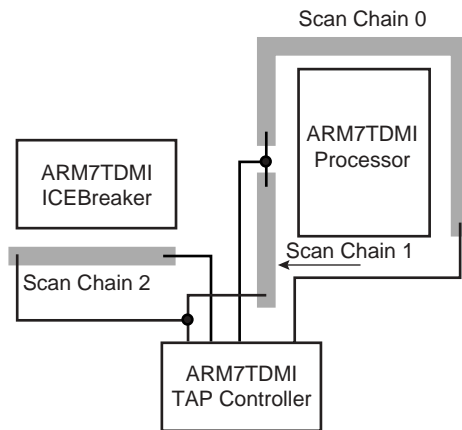
The scan cells are not fully JTAG compliant in that they do not have an Update Stage. Therefore, while data is being moved around on the scan chain, the content of the scan cell is not isolated from the output. Thus, the output from the scan cell to the core or to the external system can change on every scan clock.

This does not affect the AT91 ARM7TDMI because its internal state does not change until it is clocked. However, the rest of the system needs to be aware that every output can change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

Scan Limitations

The three scan paths are referred to as scan chain 0, 1 and 2 and are shown below in Figure 2.

Figure 2. ARM7TDMI Scan Chain Arrangement



Scan Chain 0

Scan chain 0 provides access to the entire periphery of the ARM7TDMI core, including the data bus. The scan chain functions allow inter-device testing (EXTEST) and serial testing of the core (INTEST).

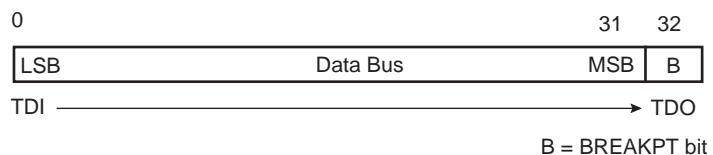
The order of the scan chain (from serial data in to data out) is:

1. Data bus bits 0 through 31
2. The control signals
3. Address bus bits 31 through 0.

Scan Chain 1

Scan chain 1 is a subset of the signals that are accessible through scan chain 0. Access to the core's data bus D[31:0], and the BREAKPT signal is available serially. There are 33 bits in this scan chain, the order being (from serial data in to out): data bus bits 0 through 31, followed by BREAKPT as shown below in Figure 3.

Figure 3. Scan Chain 1



Scan Chain 2 (ICEBreaker)

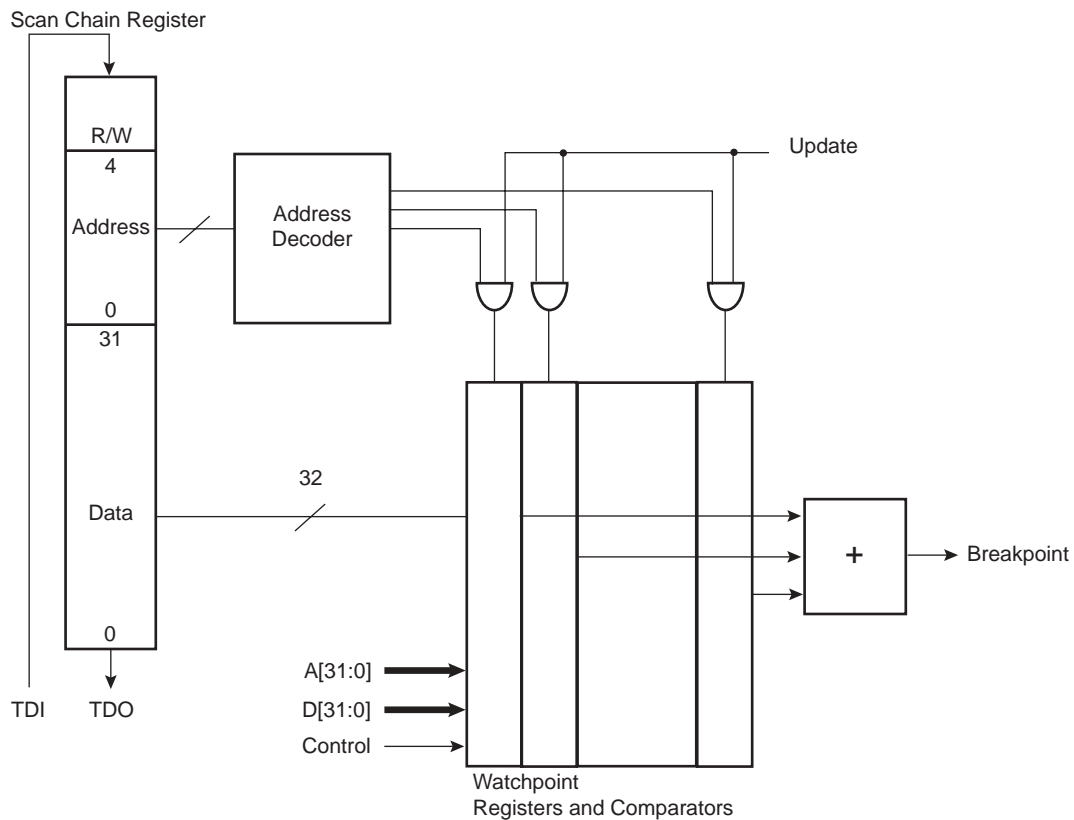
Scan chain 2 provides access to the ICEBreaker registers. The ARM7TDMI-ICEBreaker module provides integrated on-chip debug support for the ARM7TDMI processor.

ICEBreaker consists of two real time watchpoint units, together with a control and status register. One or both of the watchpoint units can be programmed to halt the execution of instructions by the ARM7TDMI via its BREAKPT signal. Execution is halted when a match occurs between the values programmed into ICEBreaker and the values currently appearing on the address bus, data bus and various control signals. Any bit can be masked so that its value does not affect the comparison.

Scan chain 2 has a length of thirty-eight bits. The order of the scan chain from TDI to TDO is:

1. Read/Write.
2. Register address bits 4 to 0.
3. Followed by data value bits 31 to 0 as shown in Figure 4.

Figure 4. ICEBreaker Block Diagram



Note: Refer to the ICEBreaker Module in the ARM7TDMI datasheet for details.

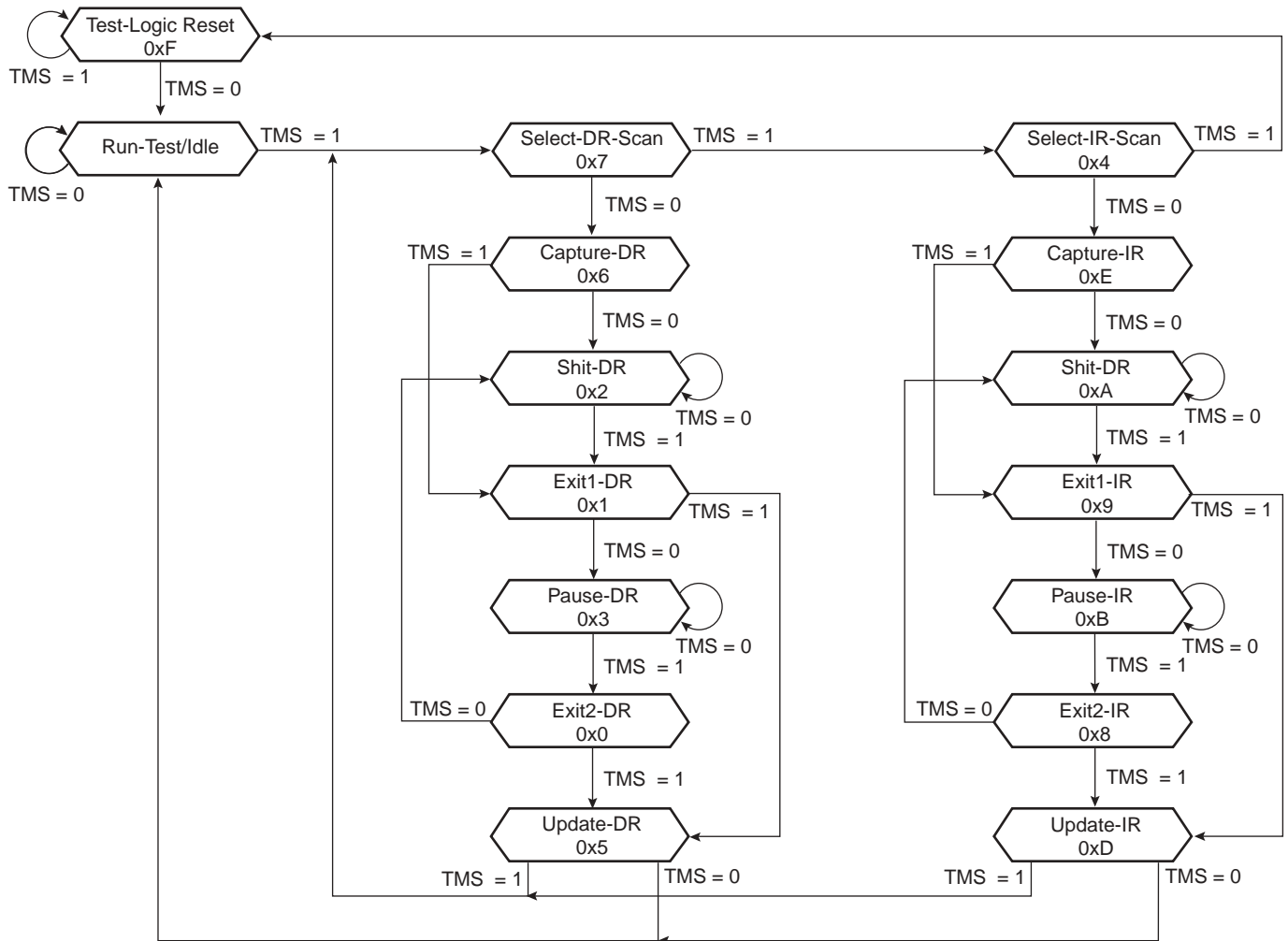
JTAG State Machine

The test and debug process is best explained in conjunction with the JTAG state machine. Figure 5 below demonstrates the state transitions that occur in the TAP controller.

All state transition occurrences of the TAP controller are based on the value of TMS at the time of a rising edge on TCK.

Note: Low state and high state of TCK must last a minimum of fifty nanoseconds each. This limits TCK to 10 MHz.

Figure 5. Test Access Port (TAP) Controller State Transitions



Instruction Register

The instruction register has a length of four bits. The three main instructions are:

1. **SCAN_N (0010):** This instruction is used to select one of the three scan chains.
2. **INTEST (1100):** This instruction places the selected scan chain in test mode. All scan cells are placed in the test mode of operation.
 - In the CAPTURE-DR state, the value of the data applied from the core logic to the output scan cells, and the value of the data applied from the system logic to the input scan cells is captured.
 - In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain via the TDO pin, while new test data is shifted in via the TDI pin.

Single-step operations are possible using the INTEST instruction.

3. **RESTART (0100):** This instruction is used to restart the processor on exit from debug state. The RESTART instruction connects the bypass register between TDI and TDO and the TAP controller behaves as if the BYPASS instruction is loaded. The processor will resynchronize back to the memory system once the RUN-TEST/IDLE state is entered.

Test Data Registers

There are six test data registers which may be connected between TDI and TDO. They are:

- Bypass Register
- ID Code Register
- Scan Chain Select Register
- Scan Chain 0
- Scan Chain 1
- Scan Chain 2.

The primary purpose of Scan Chain 1 is for debugging. Scan Chain 2 is used to program ICEBreaker. Of these six registers, the Scan Chain 1 and Scan Chain 2 Registers, as well as the Scan Chain Select Register, are the more utilizable to drive an ARM7TDMI processor.

The ID Code Register can be used when first starting to develop an application using JTAG as a driver.

Software Reset of JTAG

When starting an application, the JTAG controller might be located in any state of the JTAG state machine. Therefore, as a priority, the JTAG state machine must be in the "Test-Logic Reset" state before proceeding further.

To place the JTAG state machine in the "Test-Logic Reset" state perform the following procedure:

- Set the TMS signal
- Apply at least 5 TCK clock cycles

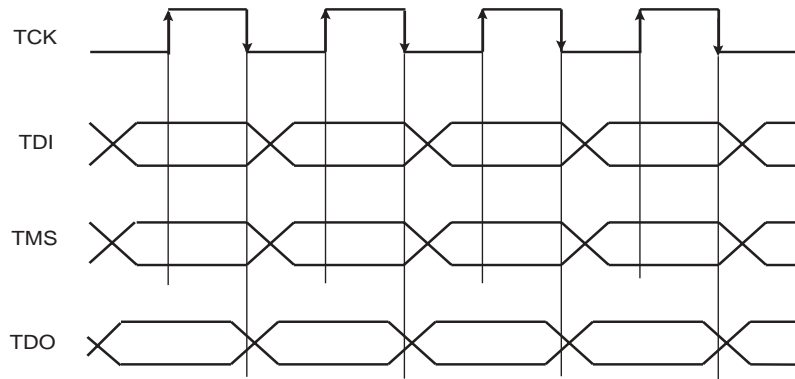
The JTAG software reset is described in the JTAG_Reset function in the jtag_opt.s assembler file.

Sending and Recovering Data

Sending and recovering data from scan chains is limited to the "Shift Instruction Register" and the "Shift Data Register".

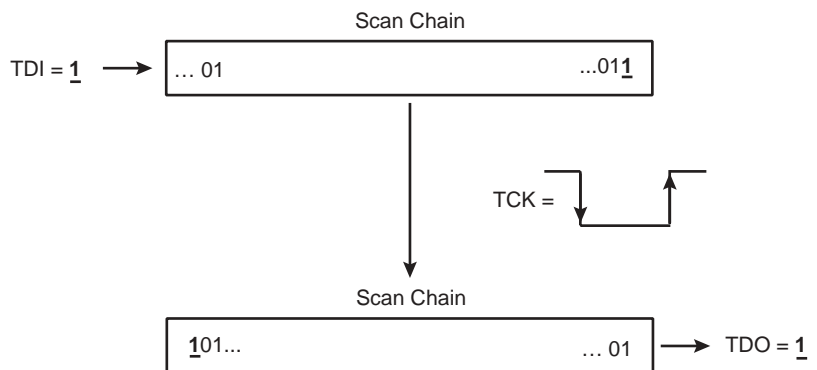
Figure 6 below shows that TDI and TMS are sampled on the rising edge of TCK and TDO transitions appear on the falling edge of TCK. Therefore, TDI and TMS must be written after the falling edge of TCK and TDO must be read after the rising edge of TCK.

Figure 6. Data Sampling and Appearance



TDO value appears forty nanoseconds maximum after the falling edge of TCK. Therefore, it is preferable to recover TDO when TCK is high. Figure 7 below demonstrates the evolution of data in a scan chain driven by TCK.

Figure 7. Data Evolution in a Scan Chain When TCK Toggles



Writing in the Instruction Register

Writing in the Instruction Register selects either a test data register or a test mode for scan chains. It initiates a new action on the ARM7TDMI processor. This concerns the correct part of the JTAG state machine, in this case all of the IR states.

To write in the Instruction Register, perform the following procedure:

Once the JTAG state machine is in "Run-Test/Idle" state, A transition to the "Shift IR" state of the JTAG state machine must be carried out. All state transitions of the TAP controller occur according to the value of TMS at the time of a rising edge on TCK. Four rising edges of TCK are needed to reach that state.

- When in "Shift IR" state, the instruction must be shifted in the instruction register. The instruction must be sent from LSB to MSB.
- The following sequence must be repeated three times:
 1. TDI = bit to send.
 2. Toggle TCK (a falling edge followed by a rising edge is advised).
- The instruction register has a length of four bits, but the last sequence, sending the MSB, differs from the first three. Once three bits are shifted into the register, MSB must be shifted in while TMS needs to be set at the same time to reach the "Exit1-IR" state when TCK toggles. It is not possible to toggle TCK without setting TMS. To

do otherwise, results in the word being shifted one bit too far into the instruction register.

- The shifted value is applied in the register in the "Update IR" state. Subsequently, an additional rising edge of TCK from the "Exit1-IR" state is required to reach the "Update IR" state.
- Once the value is applied in the register, the "Run-Test/Idle" state should be reached.

This function corresponds to the JTAG_Shift_ir function in the jtag_opt.s assembler file.

Writing in the Scan Chain Select Register

Writing in the Scan Chain Select Register changes the current active scan chain after SCAN_N has been selected as the current instruction.

The Scan Chain Select Register has a length of four bits. The writing operation is the same as writing in the instruction register except that the left part of the JTAG state machine (all DR states) is the area concerned.

Shifting in and Recovering 32-bit Words

After Scan Chain 1 (which serves primarily for debug purposes) has been selected, it is possible to shift in a 32-bit word and at the same time recover a 32-bit word. The word shifted in can be either an instruction or data, whereas the recovered word is always data.

Because some JTAG state machine states are related to an action on scan cells, they assume a significant importance:

- "Capture DR" state: The value present on the data bus is captured by the input scan cells.
- "Shift DR" state: A value is shifted in and another is shifted out of the scan cells
- "Update DR" state: The value present in the scan chain is applied to the ARM7TDMI processor.

There are 33 bits in scan chain 1, the order being (from serial data in to out): data bus bits 0 through 31, followed by BREAKPT bit. This bit, if set, indicates that the next instruction will be executed at system speed.

When the AT91 ARM7TDMI processor is stopped, it enters debug mode and can execute instructions at debug speed. There is a clock switch between the master clock (MCLK) and the debug clock (DCLK). A debug clock cycle is generated when the "Run-Test/Idle" state of the JTAG state machine is reached. A control on the clock makes a single step operation possible. The BREAKPT bit, if set, allows switching back to master clock MCLK. This is necessary if an external access is required. Effectively, at debug speed, only internal registers of the processor can be accessed because scan chains isolate the processor. So, if an external value has to be read, an external access has to be ordered and the BREAKPT bit has to be set. Consequently, the next instruction will be executed at system speed. When the execution is finished, the processor comes back into debug mode.

The comprehension of instructions executed at system speed or debug speed assumes importance for operations involving the "Complex Instruction Layer".

To write and read 32-bit words, perform the following procedure:

- Once the JTAG state machine is in "Run-Test/Idle" state, the "Shift DR" state of the JTAG state machine must be reached. While in this state, the first bit to shift in is the BREAKPT bit. Once BREAKPT bit is placed on TDI, TCK can be toggled. At this moment the instruction or the data must be shifted in and the data recovered from scan chain 1. The instruction or the data must be sent and recovered from MSB to LSB.

The following sequence must be repeated 31 times:

1. TDI = bit to send. The word is sent from MSB to LSB.
 2. Toggle TCK (a falling edge followed by a rising edge is advised)
 3. Bit to recover = TDO. The word is transferred from MSB to LSB
- To prepare the LSB bit for the instruction (or data) to be sent, TMS has to be set at the same time to reach the "Exit1-DR" state when TCK toggles. It is not advisable to toggle TCK without setting TMS. To do otherwise, results in the word being shifted one bit too far into the scan chain. Once TCK has toggled, the last bit can be recovered.
 - The shifted word is applied to the core only in the "Update DR" state. To reach the "Update DR" state, an additional rising edge of TCK from the "Exit1-DR" state is required.
 - Once value is applied to the core, the "Run-Test/Idle" state can be reached.
 - This function corresponds to the JTAG_Step function if BREAKPT bit is cleared and corresponds to the JTAG_Step_System_Speed function if BREAKPT bit is set. Both of these functions are in the jtag_opt.s assembler file.

ICEBreaker Register Map

The ICEBreaker Registers address mapping and function are shown in Table 2 below.

Table 2. Function and Mapping of ICEBreaker Registers

Address	Width	Function	Read/Write
00000	3	Debug Control	Read
00001	5	Debug Status	Read
00100	6	Debug Comms Control Register	Read
00101	32	Debug Comms Data Register	Read/Write
01000	32	Watchpoint 0 Address Value	Read/Write
01001	32	Watchpoint 0 Address Mask	Read/Write
01010	32	Watchpoint 0 Data Value	Read/Write
01011	32	Watchpoint 0 Data Mask	Read/Write
01100	9	Watchpoint 0 Control Value	Read/Write
01101	8	Watchpoint 0 Control Mask	Read/Write
10000	32	Watchpoint 1 Address Value	Read/Write
10001	32	Watchpoint 1 Address Mask	Read/Write
10010	32	Watchpoint 1 Data Value	Read/Write
10011	32	Watchpoint 1 Data Mask	Read/Write
10100	9	Watchpoint 1 Control Value	Read/Write
10101	8	Watchpoint 1 Control Mask	Read/Write

Writing in an ICEBreaker Register

After scan chain 2 has been selected, it is possible to select an ICEBreaker register and write a word in registers where writing is authorized. Scan chain 2 has a length of 38 bits.

The order of the scan chain from TDI to TDO is:

1. Read/Write
2. Register address bits 4 to 0

3. Followed by data value bits 31 to 0

To write in ICEBreaker, perform the following procedure:

- Once the JTAG state machine is in "Run-Test/Idle" state, the "Shift DR" state of the JTAG state machine must be reached. When in this state, the first bit to shift in is LSB bit of the word to write. Once LSB bit is placed on TDI, TCK can be toggled and value must be shifted in scan chain 2.
- The following sequence must be repeated 32 times:
 1. TDI = bit to send. The word is sent from LSB to MSB.
 2. Toggle TCK (a falling edge followed by a rising edge is advised)
- Now that value is shifted in scan chain 2, the ICEBreaker register address must be shifted in scan chain 2 from LSB to MSB.
- The following sequence must be repeated 5 times:
 1. TDI = bit to send. The address is sent from LSB to MSB.
 2. Toggle TCK (a falling edge followed by a rising edge is advised)
- The last bit to send is the Read/Write bit. In this instance a write is considered so the Read/Write bit must be set. Once TDI is set, TMS must be set at the same time to reach the "Exit1-DR" state when TCK toggles. It is not advisable to toggle TCK without setting TMS, to do otherwise results in the 38 bits being shifted one bit too far into the scan chain.
- The shifted 38 bits are applied to the core only in the "Update DR" state. To reach the "Update DR" state, an additional rising edge of TCK from the "Exit1-DR" state is required.
- Once value is applied to the core, the "Run-Test/Idle" state can be reached. This function corresponds to the JTAG_Write_Bkru function in the jtag_opt.s assembler file.

Reading an ICEBreaker Register

After scan chain 2 has been selected, it is possible to select an ICEBreaker register and read a word. The register must first be selected and the Read/Write bit must be cleared to indicate that a read will be performed.

To read an ICEBreaker register, perform the following procedure:

- Once the JTAG state machine is in "Run-Test/Idle" state, the JTAG state machine must be in the "Shift DR" state. When in this state, the first bit to shift in is the LSB bit of the address of the selected ICEBreaker register. Once the LSB bit is placed on TDI, TCK can be toggled and the five address bits must be shifted in scan chain 2.
- The following sequence must be repeated five times:
 1. TDI = bit to send. The word is sent from LSB to MSB.
 2. Toggle TCK (a falling edge followed by a rising edge is advised)
- The next bit to be sent is the Read/Write bit. A read is considered and the Read/Write bit must be cleared. Once TDI is cleared, TMS must be set at the same time to reach the "Exit1-DR" state when TCK is toggled. The shifted value is applied to the core only in the "Update DR" state. To reach the "Update DR" state, an additional rising edge of TCK from the "Exit1-DR" state is required.

Note: This sequence is used to select the register and to make the value appear in the scan chain when the JTAG state machine is in "Capture DR" state. Because a read is to be performed, it is not necessary to shift a 32-bit value in the scan chain.

Once the JTAG state machine is in the "Run-Test/Idle" state, the "Shift DR" state of the JTAG state machine must be reached and the value can be recovered.

- To recover the value, the procedure is as follows and must be repeated 32 times to recover the entire word:
 1. Toggle TCK (a falling edge followed by a rising edge is advised)
 2. Bit to recover = TDO. The word is transferred from LSB to MSB

Then TMS must be set to reach the "Exit -DR" state when TCK is toggled. This function corresponds to the JTAG_Read_Bkru function in the jtag_opt.s assembler file.

Complex Instructions Layer

Instructions more complex than those described in the preceding JTAG instructions are elaborated in the following paragraphs.

Selecting a Scan Chain

When selecting a scan chain the "Run Test/Idle" state should never be reached, otherwise, when in debug state, the core will not be correctly isolated and intrusive commands occur. Therefore, it is recommended to pass directly from the "Update" state" to the "Select DR" state each time the "Update" state is reached.

To select a scan chain, perform the following procedure.

- Write the SCAN_N instruction in the instruction register.
- Write the correct value in the Scan Path Select register.
- Write the INTEST instruction in the instruction register.

This is described in the JTAG_Select_SC function in the jtag-opt.s assembler file.

Determining Core State

To determine if the core is running or stopped, the Debug Status register in the ICE-Breaker must be read.

The Debug Status register has a length of 5 bits. To determine the core's state, the current status of bit 0 (**DBGACK**) and bit 3 (**nMREQ**) must first be ascertained.

When **DBGACK** is set, it indicates that the core is in debug mode. When **nMREQ** is cleared, it indicates that the ARM7TDMI has synchronized back to system speed in order to perform a memory access. To establish if a memory access has been completed, the state of both **nMREQ** and **DBGACK** must be examined. Both should be High.

To determine the core's state, perform the following procedure:

- Select the ICEBreaker scan chain.
- Read the Debug Status register.
- Examine both **nMREQ** and **DBGACK** to obtain the core's state.

This is described in the JTAG_Read_Debug_Status function in the jtag.c source file.

Core Operation Problem at Low Frequency

This issue is applicable only to a system in which the target system has a standby or low frequency clock mode (MCLK) which is much slower than the JTAG debug clock (TCK). This problem is described in the ARM Ltd. application note EAN-37 "Embedded ICE Debugging of Low-speed ARM7TDMI Systems".

Typically affected are systems that are to be actively debugged when MCLK is run below 100 kHz or where wait states can slow the effective clocking rate below this frequency. Potentially, systems with DMA controllers causing the ARM to be waited for equivalent time periods can also be affected.

Problem Summary

A synchronization problem has been identified on all ARM7 CPUs with the "I" EmbeddedICE macrocell (also referred to as "ICEBreaker") when the CPU is debugged by a

scan-clocked EmbeddedICE system (or equivalent) and when the scan clock is much faster than the CPU clock.

The only practical manifestation of this occurs in embedded designs where the CPU is stopped or clocked at a very low frequency (typically a low-power standby clock) where the scan-based debugger is able to serially shift out EmbeddedICE macrocell status register information at a rate faster than the system speed synchronization time.

This concerns particularly the AT91M42800A and the AT91M55800A chips. Both of these chips start on reset at a low frequency (32 kHz) and need the initialization of a PLL. (The AT91 microcontrollers can be similarly effected when they are in standby low-power mode.)

Low Frequency Operational Problem Description

The EmbeddedICE Interface Unit interacts with the EmbeddedICE macrocell on the target CPU by polling debug status register flags under control of the JTAG scan clock, TCK.

When the ARM7TDMI processor performs a "system speed" memory access under scan control, the instruction is scanned into the CPU then initiated by TAP controller sequencing.

It takes between one-hundred and two-hundred TCK cycles (EmbeddedICE Interface Unit firmware revision level dependent) to initiate the system-speed memory access and then poll the EmbeddedICE macrocell status to confirm completion of the operation. TCK is typically either 5 MHz or 10 MHz (EI firmware dependant), therefore when MCLK falls below 100 kHz the failure may start to occur.

To summarize, If two active cycles of MCLK have not completed (where active implies that nWAIT must have been high for at least two MCLK cycles) synchronization between debug and system clocks within the JTAG EmbeddedICE polling loop period), then erroneous status flags may be read back from the EmbeddedICE macrocell.

In a system where a debugger is used to read back memory windows (for example) to observe data, then, if the MCLK frequency is decreased below a certain ratio of MCLK to TCK, the readback data will tend to corrupt (typically to zeros). When the clock speed is increased again the memory contents are correctly read back.

Workaround

One solution to this problem is to add a waiting loop after having detected that both bits **DBGACK** and **nMREQ** are set. However, to do this can result in poor download performance.

There is no generic work around to date.

Stopping the Core

If the ARM7TDMI processor is running, it must be stopped before sending a request. The principle method to stop the core is to set watch points on any data and address activities.

To stop the ARM7TDMI processor, perform the following procedure:

- Select the ICEBreaker scan chain.
- Write 0 in the Watchpoint 0 Address Value register.
- Write 0xFFFFFFFF in the Watchpoint 0 Address Mask register to make addresses irrelevant.
- Write 0 in the Watchpoint 0 Data Value register.
- Write 0xFFFFFFFF in the Watchpoint 0 Data Mask register to make data irrelevant.
- Write 0x00000100 in the Watchpoint 0 Control Value register to enable Watchpoint 0.

- Write 0xFFFFFFFF7 in the Watchpoint 0 Control Mask register to detect only the fetch instruction.
- Wait that the core enters debug mode by reading the Debug Status register.
- Write 0 in the Watchpoint 0 Control Value register to disable Watchpoint 0.
- A JTAG reset is advised once ARM7TDMI is stopped.

This operation is described in the JTAG_Stop function in the jtag.c source file.

Pushing an Instruction into the Pipeline

To push an instruction into the pipeline, perform the following procedure:

- Select the Debug scan chain (scan chain 1)
- Write the 32-bit code corresponding to the instruction to execute.

This operation is described in the JTAG_Execute function in the jtag.c source file.

The "nop" Instruction

A "nop" instruction is carried out by using the assembler instruction **MOV r0, r0** The code corresponding to this instruction is 0xE1A00000.

The "nop" instruction is used to push instructions into the ARM7TDMI three-stage pipeline and prevents the core from falling into an undesired mode by fetching a not recognized instruction.

When the "Run-Test/Idle" state of the JTAG state machine is reached, the instruction enters the pipeline. Subsequently, DCLK clock cycles appear in that state and two other instructions must be pushed in the pipeline before the "nop" is executed.

Reading a Register

Reading a register is achieved by pushing an instruction (**STR R[index], [R14]** for example) into the pipeline. This instruction places the value on the data bus. The code corresponding to this instruction is 0xE58E0000. The selected register is indicated from bit 12 to bit 15 and must be added to the code.

Reading a register can be executed at debug speed because no external memory access is needed.

Note: It is extremely important to understand and perform exactly the procedures specific to the desired instruction(s) presented in the "Instruction Cycle Operations" chapter of the ARM7TDMI datasheet.

Table 3. Store Register Cycle Operations

Cycle	Address	Data	Comment
1	X ⁽¹⁾	X ⁽¹⁾	Instruction In Fetch Stage of pipeline
2	X ⁽¹⁾	X ⁽¹⁾	Instruction in Decode Stage of pipeline
3	pc + 2L ⁽²⁾	(pc + 2L) ⁽²⁾	Instruction in Execute Stage of pipeline
4	Alu	Rd	Caution. Data appears on falling edge of DCK

- Notes: 1. X = don't care
2. L indicates the instruction length (4 bytes in ARM mode).

Once the Instruction Cycle Operations are defined, it is possible to perform the following programming procedure:

- Push the **STR R[index], [R14]** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.

- Push a "nop" instruction into the pipeline. The instruction is now in the Execute stage of the pipeline.
- Push a "nop" instruction into the pipeline. The data will appear on the data bus on falling edge of DCK. A DCK clock cycle is performed when the "Run-Test/Idle" state of the JTAG state machine is reached.
- Recover the 32-bit word.

This operation is described in the JTAG_Read_Register function in the jtag.c source file.

Writing in a Register

Writing a register is achieved by pushing an instruction (**STR R[index], [R14]** for example) into the pipeline. This instruction places the value on the data bus. The code corresponding to this instruction is 0xE59E0000. The selected register is indicated from bit 12 to bit 15 and must be added to the code.

Table 4. Load Register Cycle Operations

Cycle	Address	Data	Comment
1	X ⁽¹⁾	X ⁽¹⁾	Instruction In Fetch Stage of pipeline
2	X ⁽¹⁾	X ⁽¹⁾	Instruction in Decode Stage of pipeline
3	pc + 2L ⁽²⁾	(pc + 2L) ⁽²⁾	Instruction in Execute Stage of pipeline
4	alu	Data to write	Push the data to write on data bus
5	pc + 3L ⁽²⁾	X ⁽¹⁾	Data is written in the selected register
6	X ⁽¹⁾	X ⁽¹⁾	Only if the selected register is PC
7	X ⁽¹⁾	X ⁽¹⁾	Only if the selected register is PC

- Notes:
1. X = don't care
 2. L indicates the instruction length (4 bytes in ARM mode).

Once the Instruction Cycle Operations are defined, it is possible to perform the following programming procedure:

- Push the **LDR R[index], [R14]** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Execute stage of the pipeline.
- Push the 32-bit word to write. The data must be present on the data bus before rising edge of DCK. A DCK clock cycle is performed when the "Run-Test/Idle" state of the JTAG state machine is reached.
- Push a "nop" instruction into the pipeline. This cycle allows the word to be written in the selected register.

If the selected register is the program counter (PC):

- Push a "nop" instruction into the pipeline.
- Push a second "nop" instruction into the pipeline.

This operation is described in the JTAG_Write_Register function in the jtag.c source file.

Reading a 32-bit Word in External Memory

Reading a word in external memory differs from other reads because several instructions are needed. The primary instruction is **LDR R1, [R0], #4** (for example). The address of the word to read is written first in the R0 register. Reading a word in memory requires an external memory access. Therefore, the instruction must be executed at system speed to prevent the read value from being the value previously present in the R1 register. The code corresponding to this instruction is 0xE4901004.

When an instruction requires an external memory access, there is no need to know the "Instruction Cycle Operations". The instruction is executed at system speed and the ARM7TDMI core re-enters debug mode when the execution is finished.

To read a word in external memory perform the following procedure:

- Read the R0 register and save the recovered value.
- Read the R1 register and save the recovered value.

Note: This is essential because both registers are used to read a 32-bit word in external memory.

- Write the selected address in the R0 register.
- Push a "nop" instruction into the pipeline to clean it.
- Push a "nop" instruction into the pipeline with **BREAKPT** set to indicate that the next instruction must be executed at system speed.
- Push the **LDR R1, [R0], #4** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Select the **RESTART** instruction in the instruction register. The instruction is executed at system speed. The core synchronizes back to MCLK.
- Wait the instruction until finished by reading the Debug Status Register in the ICEBreaker. **DBGACK** and **nMREQ** bits must be set.

The core should be stopped at this point.

- Recover the data by reading the R1 register.
- Restore the R0 register by writing the saved value in that register.
- Restore the R1 register by writing the saved value in that register.

This operation is described in the JTAG_Read_Memory function in the jtag.c source file.

Reading the CPSR Register to Determine Current Mode

To determine the current mode, read the CPSR register by pushing the instruction **MRS R0, CPSR** (for example) into the pipeline. This instruction stores the result in the R0 register. The code corresponding to this instruction is 0xE10F0000.

Reading the CPSR register can be executed at debug speed because no external memory access is needed.

Table 5. MRS Cycle Operations

Cycle	Address	Data	Comment
1	X ⁽¹⁾	X	Instruction in Fetch Stage of pipeline
2	X	X	Instruction in Decode Stage of pipeline
3	X	X	Instruction in Execute Stage of pipeline

Note: 1. X = don't care

To read the CPSR register perform the following procedure:

- Read the R0 register and save the recovered value.
- Push the MRS R0, CPSR instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Execute stage of the pipeline.
- Recover the value by reading the R0 register.
- Restore the R0 register by writing the saved value in that register.

This operation is described in the JTAG_Read_CPSR function in the jtag.c source file.

Writing in CPSR Register to Set a Mode

Writing in CPSR register is achieved by pushing the instruction **MSR cpsr_cxsf, R0** into the pipeline. This instruction requires that the value be stored in the R0 register. The code corresponding to this instruction is 0xE12FF000.

Writing in CPSR register can be executed at debug speed because no external memory access is needed.

Table 6. MSR Cycle Operations

Cycle	Address	Date	Comment
1	X ⁽¹⁾	X	Instruction in Fetch Stage of pipeline
2	X	X	Instruction in Decode Stage of pipeline
3	X	X	Instruction in Execute Stage of pipeline

Note: 1. X = don't care

To write in CPSR register perform the following procedure:

- Read the R0 register and save the recovered value.
- Write the new CPSR value in the R0 register.
- Push the **MSR cpsr_cxsf, R0** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Execute stage of the pipeline. The value is now written in CPSR register.
- Restore the R0 register by writing the saved value in that register.

This operation is described in the JTAG_Write_CPSR function in the jtag.c source file.

Storing Multiple Values in Memory

This function can be used to copy a program into memory.

Two phases are necessary to store multiple values in memory.

1. In the first phase, a group of data is copied into the core internal registers.
2. The second phase consists in storing these data in memory by using a store multiple instruction.

Loading Multiple Data in Internal Registers

Loading multiple data in internal registers is achieved by pushing the instruction **LDMIA LR, {R0-Rx}** in the pipeline. This instruction provides storing of up to 16 data, from R0 to R15, but it is recommended not to go over 14 data in any single operation. However, the

load multiple function can be used to restore the context and, in this case, the 16 registers can be used in one operation.

The code corresponding to this instruction is 0xE89E0000. The selected registers are indicated from bit 0 to bit 15, each bit represents a register. Bit 0 represents the R0 register and the bits continue sequentially to R15. This information must be added to the code.

This function can be executed at debug speed because no external memory access is needed.

Table 7. Load Multiple Cycle Operations

Cycle	Address	Data	Comment
1	X ⁽¹⁾	X	Instruction in Fetch Stage of pipeline
2	X	X	Instruction in Decode Stage of pipeline
3	pc + 2L ⁽²⁾	(pc + 2L)	Instruction in Execute Stage of pipeline
4	alu	(data)	Push the first word to write on data bus
	alu++	(data++)	Push the next word to write on data bus
⁽³⁾ N+3	alu++	(data++)	Push the next word to write on data bus
N+4	alu++	pc'	Push the new PC value if necessary
N+5	X	X	Last data is written in the register
N+6	X	X	Only if PC is selected
N+7	X	X	Only if PC is selected

- Notes:
1. X: don't care
 2. L: Indicates the instruction length(4 bytes in ARM mode)
 3. N: Number of Registers

To write multiple words in internal registers perform the following procedure:

- Push the **LDMIA LR, {R0-Rx}** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Execute stage of the pipeline.
- Push as many words as defined in the instruction.
- Push a "nop" instruction into the pipeline. This cycle allows the last word to be written in the last register.

If a value is written in the PC register:

- Push a "nop" instruction into the pipeline.
- Push a second "nop" instruction into the pipeline.

This operation is described in the JTAG_Load_Multiple function in the jtag.c source file.

Storing Multiple Data in Memory

Storing data in memory is achieved by pushing instruction **STMIA R14!, {R0-Rx}** into the pipeline. This instruction allows storing up to 14 data at the same time from R0 to R13 at the address indicated in R14 register. The code corresponding to this instruction is 0xE88E0000. The selected registers are indicated from bit 0 to bit 15 with each bit representing a register. Bit 0 represents the R0 register and the bits continue sequentially to R14. This must be added to the code.

Writing multiple words in memory requires external memory accesses. Consequently, the instruction must be executed at system speed.

To write multiple words in memory perform the following procedure:

- Write the selected address in the R14 register.
- Push a "nop" instruction into the pipeline to clean it.
- Push a "nop" instruction into the pipeline with **BREAKPT** set to indicate that the next instruction has to be executed at system speed. The "nop" instruction cleans the pipeline.
- Push the **STMIA R14!, {R0-Rx}** instruction code into the pipeline. The instruction is now in the Fetch stage of the pipeline.
- Push a "nop" instruction into the pipeline. The instruction is now in the Decode stage of the pipeline.
- Select the **RESTART** instruction in the instruction register. The instruction is executed at system speed. The core synchronizes back to MCLK.
- Wait that the instruction is finished by reading the Debug Status Register in the ICEBreaker. Both bits **DBGACK** and **nMREQ** have to be set.

The core should be stopped at this point.

This operation is described in the `JTAG_Store_Multiple` function in the `jtag.c` source file.

Restarting the ARM7TDMI Core

Restarting the ARM7TDMI core can be difficult to do because the PC register value must be controlled. Essentially, the core must start at the correct address.

Note: When a branch instruction is in the pipeline with a prior instruction set in the **BREAKPT** bit, the core synchronizes back to MCLK and does not return to debug mode. This condition must be met to restart the core.

To restart the core, perform the following procedure:

- Use the Load Multiple function to restore or to set the context. The value indicated in the PC must be the restart address.
- Push a "nop" instruction into the pipeline with the **BREAKPT** set to indicate that the next instruction has to be executed at system speed.
- Push a **branch [PC-X]** instruction into the pipeline. X represents the gap between the moment the address has been stored and the present moment. X is a 4 multiple number.
- Select the **RESTART** instruction in the instruction register.

This is described in the `JTAG_Go` function in the `jtag.c` source file.

Application Example: Flash Up-loader

By using the instructions described in the earlier sections of this document, it is possible to implement an application such as a flash up-loader. The flash up-loader is capable of reprogramming an AT91 evaluation board target by writing an application in its flash memory.

The supported AT91 evaluation board targets are:

- the AT91EB40 (equipped with Atmel Flash AT29LV1024)
- the AT91EB40A, the AT91EB42, the AT91EB55 and the AT91EB63 (each equipped with Atmel Flash AT49BV).

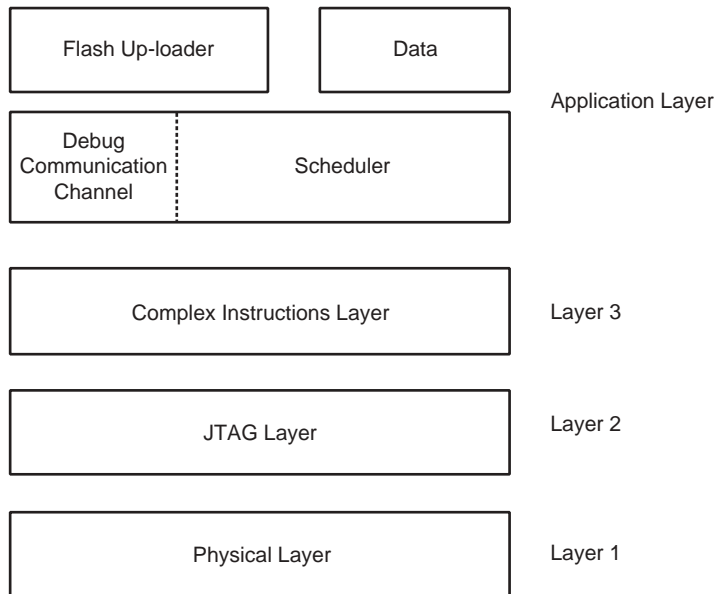
The main features of the flash up-loader are:

- Automatic detection of the AT91 Evaluation Board to which it is connected by checking the Chip ID Register.

- Loading the boot program into flash, the ROM Angel monitor and an application corresponding to the target.
- Use of the Ice Debug communication channel for data transfers

General Synoptic

Figure 8. General Synoptic



Data Flow

Figure 9. Load Target Code

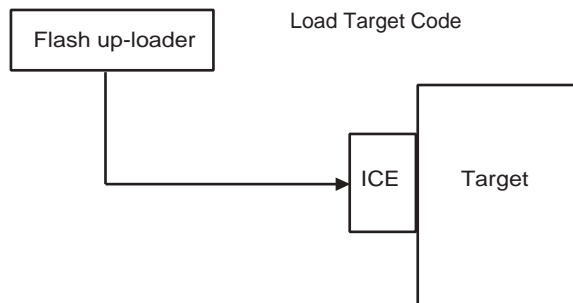
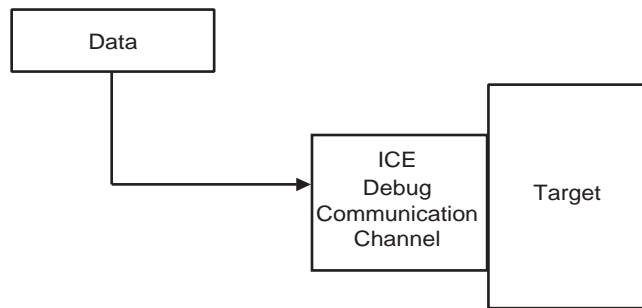


Figure 10. Load Target Data



Application Description

The AT91EB40A Flash Up-loader operates in several phases as described below:

- The AT91EB40A host stops the core of the target board.
- The AT91EB40A host reads the Chip ID register at address 0xFFFF0000 of the target board to identify the target.
- Once the target board is identified, the AT91EB40A initializes the peripherals needed such as PLL for the AT91EB42 and the AT91EB55 Evaluation Boards and EBI for all target boards.
- The AT91EB40A host sets the target in Supervisor Mode by writing in the CPSR register and downloads a program to identify the target board flash into the internal RAM of the target board. This is accomplished by using the LDM and STM functions.
- Once the program is loaded in internal RAM, the AT91EB40A host sends a branch instruction at the download address to the target board. The core restarts and the downloaded program is executed. The host obtains the manufacturer and device identification codes through the ICEBreaker Communication Channel register of the target board.
- The AT91EB40A host stops the core of the target board and verifies that the core is still in Supervisor Mode by reading the CPSR register and downloads a specific flash write program into the internal RAM depending on the flash type of the target board. This program is used to program the identified flash. Once the program is loaded in internal RAM, the AT91EB40A host sends a branch instruction at the download address to the target board. The core restarts and the downloaded program starts. The host sends the data to be stored through the ICEBreaker Communication Channel register of the target board.⁽¹⁾

Note: 1. This step is repeated three times because three different programs listed below need to be stored in flash:

- The boot corresponding to the target board.
- The ROM Angel monitor corresponding to the target board.
- An application corresponding to the target board.

All programs must be stored in flash of the AT91EB40A host. These include:

- Boot programs
- ROM monitor programs
- Applications
- Flash identification programs
- Flash programming programs

The mapping is defined in the eb_desc.c source file and the application is defined in the flash_uploader_appl.c source file.

All source code relevant to this application note can be downloaded from the Atmel web site. <http://www.atmel.com>





Document Details

Title AT91 ARM Thumb Application Note: Using an AT91EB40A Evaluation Board to Control an AT91 ARM7TDMI Core Via the ARM JTAG-ICE Interface.

Literature Number 2668

Revision History

Version A **Publication Date:** 14 January, 2003



Atmel Headquarters

Corporate Headquarters

2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe

Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia

Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan

9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory

2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers

2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards

Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive

Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/ High Speed Converters/RF Datacom

Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail

literature@atmel.com

Web Site

<http://www.atmel.com>



© Atmel Corporation 2003.

Atmel Corporation makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in Atmel's Terms and Conditions located on the Company's web site. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of Atmel are granted by the Company in connection with the sale of Atmel products, expressly or by implication. Atmel's products are not authorized for use as critical components in life support devices or systems.

ATMEL®, is the registered trademark of Atmel. ARM®, ARM7TDMI®, ARM®Thumb® and ARM Powered® are the registered trademarks of ARM Ltd.

Other terms and product names may be the trademarks of others.



Printed on recycled paper.