

---

## Technical Notes

# Intel XScale Family On-Chip Emulation

## Contents

Contents.....	1
1 Introduction .....	2
2 Emulation Options.....	3
2.1 Hardware Options .....	3
2.2 Initialization Sequence .....	4
2.3 JTAG Scan Speed .....	6
3 CPU Options.....	7
3.1 General Options .....	7
3.2 Debugging Options .....	8
3.3 Advanced Options.....	9
3.4 Exceptions .....	10
4 Real-Time Memory Access .....	12
5 Access Breakpoints .....	12
6 Trace Configuration.....	13
7 Getting Started.....	14
8 Troubleshooting.....	21
9 Notes.....	21

# 1 Introduction

The JTAG interface offers all basic debug functions, based on which a debugger is implemented in winIDEA: Read and Write Memory, Read and Write Registers, Run and Stop. Single step is not supported by on-chip hardware, so it must be implemented by the debugger on a higher level. Two hardware execution breakpoints are provided on-chip, one of which is usually reserved for winIDEA for single-stepping; the other one is available to the user. If the code is loaded into RAM, unlimited number of software breakpoints can be used. The reserved hardware breakpoint can be freed in the Debug/Debug Options/Debugging menu by unchecking the 'Reserve one breakpoint for high level debugging' option.

---

Note: If 'Reserve one breakpoint for high level debugging' option is unchecked and all hardware breakpoints are used, this disables high-level debugging features. This means that single-stepping is not possible. If a breakpoint is still available, the high-level debugging and stopping will still be operational.

---

Processors of Intel XScale family are based on a common processing core that is compliant with ARM Version 5TE instruction set and programming model. XScale core contains on-chip debug interface that is accessible via a standard JTAG port usually using a standard 20-pin ARM JTAG connector.

## Debug Features:

- Two hardware instruction breakpoints
- Two hardware data breakpoints which can be used independently or in combination
- Unlimited number of software breakpoints
- Instruction cache, MMU and data cache support
- Real-time memory access
- THUMB support
- Fast FLASH programming
- Program execution on-chip trace

## ARM THUMB

The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16-bit long and have corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb instructions operate with standard ARM register configuration, allowing excellent interoperability between ARM and Thumb states.

On execution, 16-bit Thumb instructions are transparently decompressed to full 32-bit ARM instructions in real-time, without performance loss.

Thumb code is typically 65% of the size of ARM code and provides 160% of the performance of ARM code when running on a processor connected to a 16-bit memory system. Thumb therefore is an advantage in applications with restricted bandwidth, where code density is important. The availability of both 16-bit Thumb and 32-bit ARM instruction sets gives designers the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications.

Switching from native ARM 32-bit instruction set to 16-bit Thumb and back represents some overhead for the application from the aspect of the overall performance. In the real application, the executed Thumb code should be big enough that the increase in performance due to Thumb instruction set use overcomes the loss in performance due to necessary switch from 32-bit instruction set to Thumb instruction set and switch back to native 32-bit instruction set at the end of the Thumb code.

## THUMB Code Debugging

The debugging can be performed as normal. When the THUMB code is being stepped, the Data in the Code window is 16-bit long; otherwise it is 32-bit long. Not all CPU registers are available in THUMB mode, although they can be seen in the SFR window. See the THUMB section of the ARM7 manual for more information.

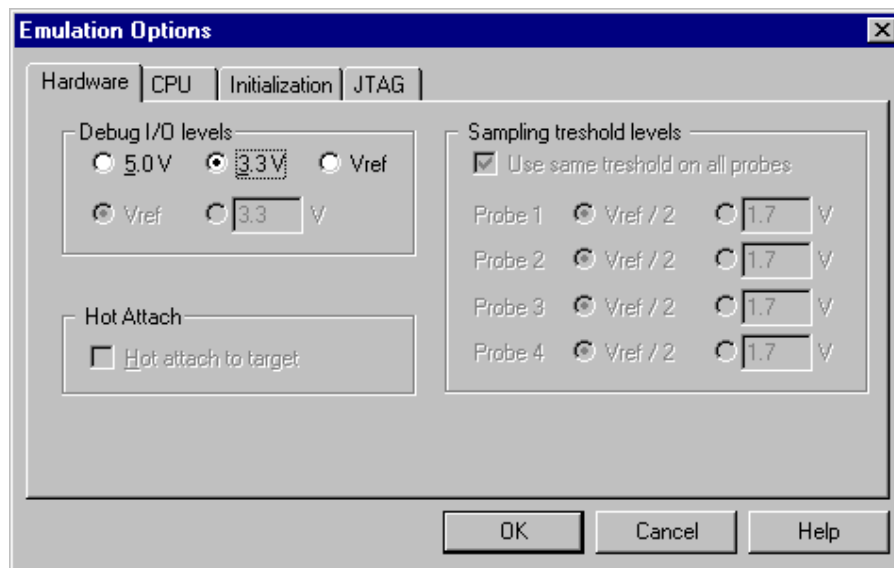
## Supported CPUs

winIDEA supports all CPUs based on the Intel XScale core. Several microcontrollers have already been implemented, also with special function register (SFR) information. The special function registers can be implemented for any microcontroller on request, only the SFR specification must be supplied. Also user defined custom SFR definitions can be added, see the Hardware User's Manual for more information.

Check with iSYSTEM for the latest list of supported CPUs.

## 2 Emulation Options

### 2.1 Hardware Options



*Emulation options, Hardware pane*

### **Debug I/O levels**

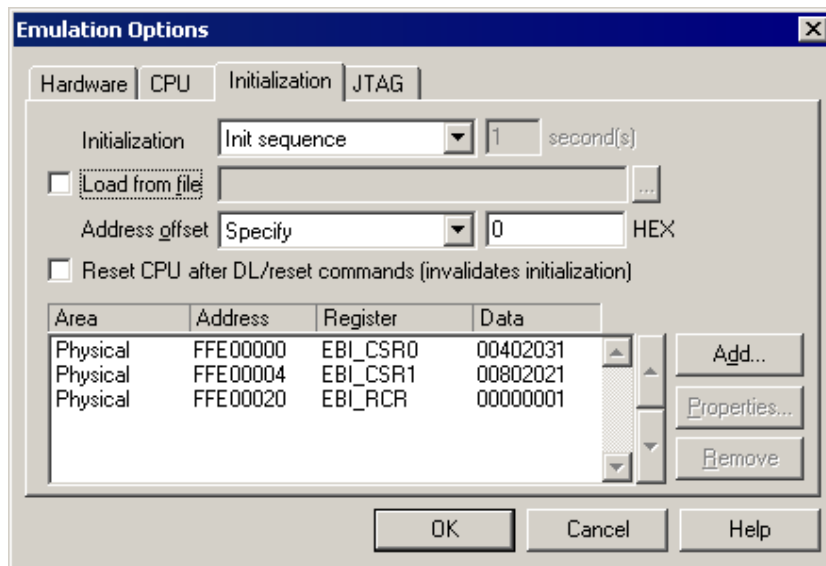
The development system can be configured in a way that the debug JTAG signals are driven at 3.3V, 5V or target voltage level (Vref). When 'Vref' Debug I/O level is selected, a voltage applied to the belonging reference voltage pin on the target debug connector is used as a reference voltage for driving the debug JTAG signals. Make sure that the target reference voltage pin is connected when 'Vref' Debug I/O level is selected otherwise the emulation will fail.

## 2.2 Initialization Sequence

Before flash programming or download can take place, the user must ensure that the memory is accessible. This is very important since there are many applications using memory resources (e.g. external RAM, external flash), which are not accessible after the CPU reset. In that case, the debugger must execute after the CPU reset a so called initialization sequence, which configures necessary CPU chip selects and then download or flash programming can actually take place. The user must set up the initialization sequence depending on the application.

The initialization sequence can be set up in two ways:

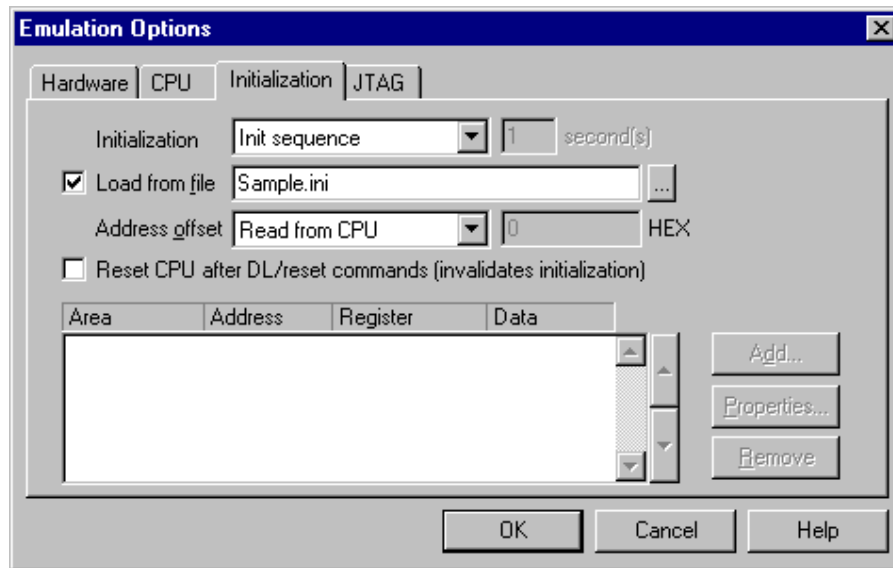
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA Emulation Options -> Initialization page.



2. winIDEA can load an initialization sequence from a text file with .ini extension. The file must be formatted according to the syntax specified in the appendix in the hardware user's guide.

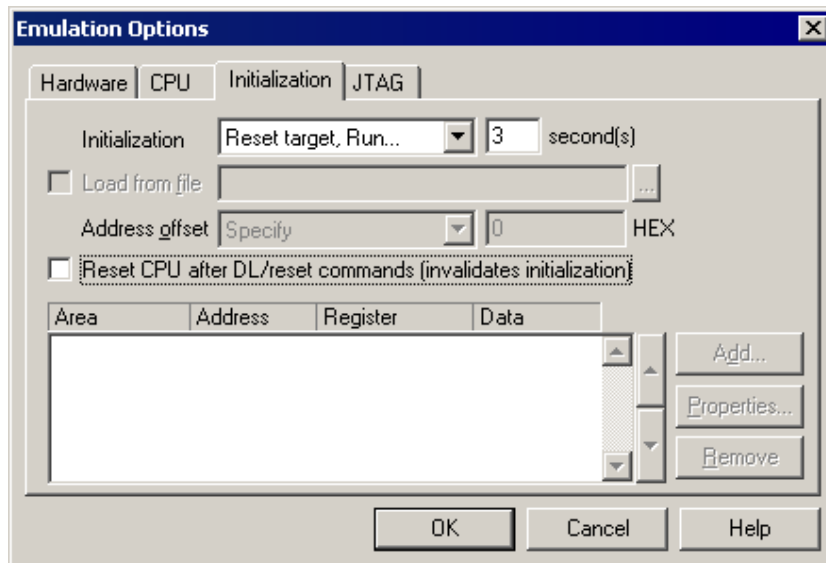
Excerpt from the Sample.ini file:

```
S EBI_CSR0 L 0x00402031 // CS0 - ext. flash, 4 wait states
S EBI_CSR1 L 0x00802021 // CS1 - ext. SRAM
S EBI_RCR L 0x00000001 // remap internal RAM
```

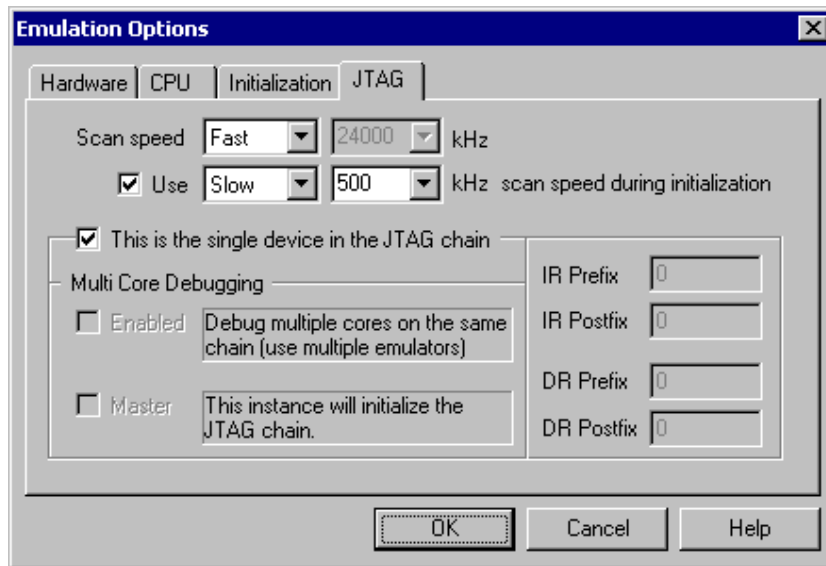


The advantage of the second method is that you can simply share your .ini file among different workspaces and users. Additionally, you can easily comment out some lines while debugging the initialization sequence itself.

There is also a third method, which is suitable in some situations. The user can initialize the CPU by resetting the target system, letting it run for a certain number of seconds and then stopping it. This executes target's firmware, which can take care of system and peripheral initialization. This can be done using 'Reset and run for X sec' option.



## 2.3 JTAG Scan Speed



*JTAG Scan Speed definition*

### *Scan speed*

The JTAG chain scanning speed can be set to:

- Slow - long delays are introduced in the JTAG scanning to support the slowest devices. JTAG clock frequency varying from 1 kHz to 2000 kHz can be set.
- Fast – the JTAG chain is scanned with no delays.
- Burst – provides the ability to set the JTAG clock frequency varying from 4 MHz to 100 MHz.
- Burst+ - provides the ability to set the JTAG clock frequency varying from 4 MHz to 100 MHz
- RTCK - Adaptive RTCK clocking for ARM

Slow and Fast JTAG scanning is implemented by means of software toggling the necessary JTAG signals. Burst mode is a mixture of software and hardware based scanning and should normally work except when the JTAG scan frequency is an issue that is when the JTAG scan frequency used by the hardware accelerator is too high for the CPU. In general, selecting an appropriate scan frequency usually depends on scan speed limitations of the CPU. In Burst+ mode, complete scan is controlled by the hardware accelerator, which poses some preconditions, which are not met with all CPUs. Consequentially, Burst+ mode doesn't work for all CPUs. Burst and Burst+ are not supported on iONE debug tool.

RTCK speed mode is available for ARM family only and is intended for targets which use widely varying system clock during a debug session. For example, if the CPU switches to different power modes and changes system clocks, the debugger will be able to maintain synchronization with on-chip debug interface even at much slower clock. The target CPU needs to provide RTCK synchronization signal, which must be available on pin 11 on standard 20-pin ARM JTAG debug connector. RTCK clock option is available for all development systems except for older iC3000 ARMx iCARD based development system. Due to extra synchronization, top speed using "RTCK" mode is about half as fast as "Fast" mode.

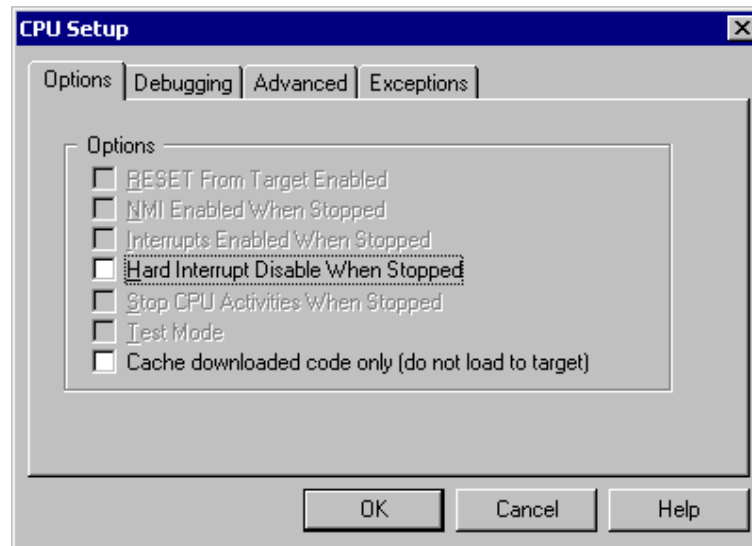
In general, Fast mode should be used as a default setting. If the debugger works stable with this setting, try Burst or Burst+ mode to increase the download speed. If Fast mode already fails, try Slow mode at different scan frequencies until you find a working setting.

### ***Use – Scan Speed during Initialization***

Some target systems require use of a slower scan speed during initialization when CPU is not yet running at full clock speed. After CPU clock is raised to full operating frequency (PLL engaged) higher scan speeds can be used. In such a case, this option should be used with appropriate slow scan frequency.

## **3 CPU Options**

### **3.1 General Options**



*XScale Family Debugging Options*

#### ***Hard Interrupt Disable When Stopped***

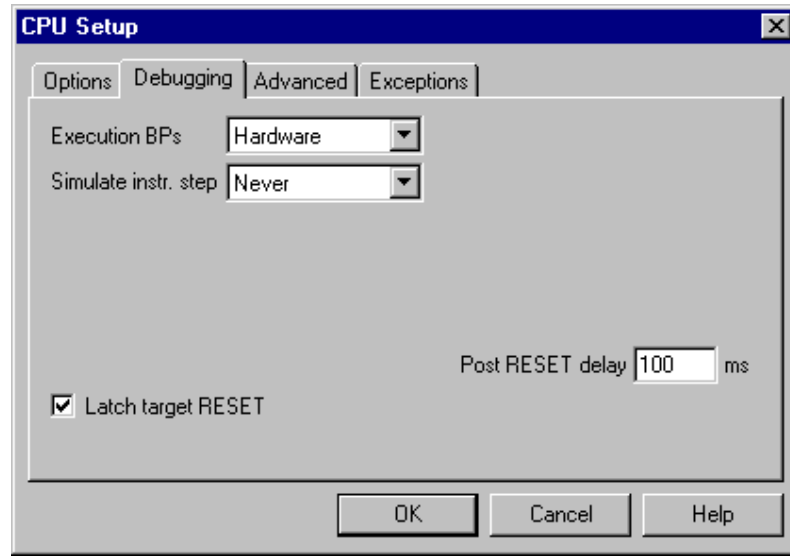
When this option is checked, interrupts will be enabled immediately after program execution resumes. Otherwise, the CPU must execute a couple of instructions before returning to the program to determine whether interrupts were enabled when the CPU was stopped. These extra instruction executions can prevent task preemption when an interrupt is already pending.

#### ***Cache downloaded code only (do not load to target)***

When this option is checked, the download files will not propagate to the target using standard debug download but the Target download files will.

In cases, where the application is previously programmed in the target or it's programmed through the flash programming dialog, the user may uncheck 'Load code' in the 'Properties' dialog when specifying the debug download file(s). By doing so, the debugger loads only the necessary debug information for high level debugging while it doesn't load any code. However, debug functionalities like ETM and Nexus trace will not work then since an exact code image of the executed code is required as a prerequisite for the correct trace program flow reconstruction. This applies also for the call stack on some CPU platforms. In such applications, 'Load code' option should remain checked and 'Cache downloaded code only (do not load to target)' option checked instead. This will yield in debug information and code image loaded to the debugger but no memory writes will propagate to the target, which otherwise normally load the code to the target.

## 3.2 Debugging Options



*XScaleFamily Debugging Options*

### **Execution Breakpoints**

#### *Hardware Breakpoints*

Hardware breakpoints are breakpoints that are already provided by the CPU. The number of hardware breakpoints is limited to two. The advantage is that they function anywhere in the CPU space, which is not the case for software breakpoints, which normally cannot be used in FLASH memory, non-writeable memory (ROM) or self-modifying code. If the option 'Use hardware breakpoints' is selected, only hardware breakpoints are used for execution breakpoints.

Note that the debugger, when executing source step debug command, uses one breakpoint. Hence, when all available hardware breakpoints are used as execution breakpoints, the debugger may fail to execute debug step. The debugger offers 'Reserve one breakpoint for high-level debugging' option in the Debug/Debug Options/Debugging' tab to prevent this from happening. By default this option is checked and the user can uncheck it anytime.

#### *Software Breakpoints*

Number of available hardware breakpoints often proves to be insufficient. Debugger can use unlimited software breakpoints to work around this limitation.

When a software breakpoint is used, the program first attempts to modify target code by placing a break instruction into the code. If setting software breakpoint fails, a hardware breakpoint is used instead if one is available.

#### *Simulate instr. step*

'Never' is selected per default. When run or source step debug command is executed from a BP location, the debugger first clears BP, executes single step, sets back the original BP and then resumes the application. All this is done in background hidden from the user. Since setting and clearing software flash breakpoint is time consuming, a new approach was introduced, which simulates the first instruction at breakpoint address without requiring clearing and setting the software flash breakpoint. Thereby, the user can select 'FLASH SW BP' in order to speed up the debugging. If the option yields erroneous behavior, set back to the default setting.

### ***Latch target RESET***

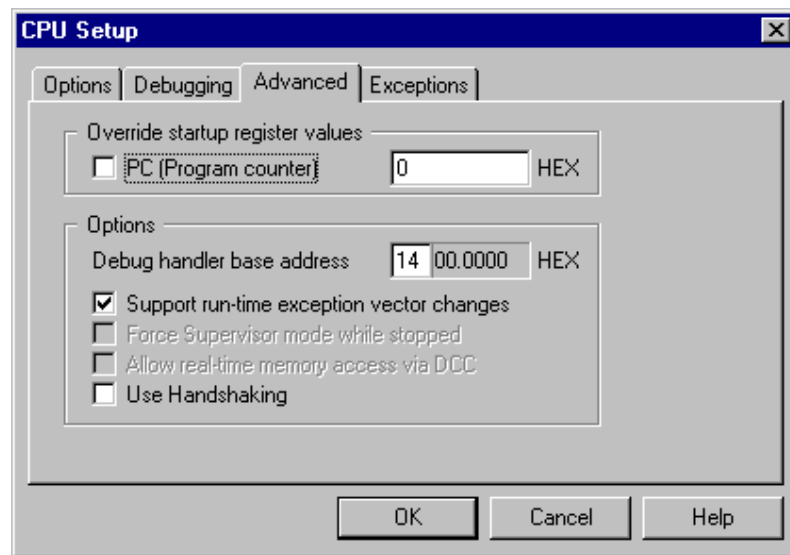
When the option is checked, the debugger latches active signal on the target reset line, processes it, resets the complete system and resumes the application. This yields a delay between the target reset and restart of the application from reset. If this delay is not acceptable for a specific application, the option should be unchecked. An example is the application where the CPU is periodically set into a power save mode and is then waken up e.g. every 6ms by an external reset circuit. In such case, a delay introduced by the debugger would yield application not operating properly. Per default, the option is checked.

When the option is unchecked, it may happen that the debugger does not detect the target reset although the CPU gets reset. The debugger polls the CPU status ~3 times per second while target reset can occur in between.

### ***Post RESET Delay***

Typically, the on-chip debug module is reset concurrently with the CPU. After the CPU reset line is released from the active state, the on-chip debug module requires some time (delay) to become operational. The default delay value normally allows the debugger to gain the control over the CPU. If first debug connection fails try different delay values to establish debug connection.

## **3.3 Advanced Options**



*XScale Advanced Options*

### ***Override startup register values***

This option overrides default Program Counter reset value. This means when the CPU is reset, the Program Counter will be set to the value provided by this option.

### ***Debug handler base address***

On XScale, a debug handler is used for performing debug control and operations. Debug handler runs from internal mini instruction cache which is reserved specifically for debugging purposes. This mini I-cache is 2KB in size. Application code is never cached in this mini I-cache, however, if application makes instruction fetches from addresses that are mapped in the mini I-cache, then code will be fetched from mini I-cache. That is why debug handler code that resides in the mini I-cache must be mapped to addresses that do not overlap with application code space (but it doesn't hurt if it is mapped to application's data area, because no instruction fetches are expected to be made there)

This option is provided to control the base address where debug handler is mapped to. Only the most significant byte can be specified.

### *Support run-time exception vector changes*

- Unchecked (OFF) – debugger does not support changes of exception vectors while target application is running. This means that if target application goes into running, at some point changes vectors while running and then hits an exception, the emulation will fail. Having this option unchecked brings no performance penalty to exception handling and exception handler invocation and execution can be correctly recorded by on-chip trace. Leaving this option OFF is most suitable for applications that never change exception vectors.
- Checked (ON) – Debugger handles target applications that change exception vectors while running. XScale debugger by design always has to set up its own exception vector table in the debug-dedicated mini instruction cache. Whenever an exception is raised, exception vector is always fetched from debug-dedicated mini instruction cache, not from the application's vector table in system memory. However, a running application can change its own exception vector table(s) in memory at any point in time and the debugger doesn't know when that happens. Turning this option ON configures the debugger to correctly handle these vector changes and makes sure the application's exception handlers are invoked. However, there is a performance penalty of 15-20 instructions in this case. Also, trace recording is affected by this option, as exception handler execution can not be traced correctly in most situations.

Turning this option ON is suitable for target systems that change exception vectors during runtime.

This option can be toggled ON/OFF at any point during debug session. For example, if target application does change exception vectors at runtime (only once at startup though) and performance penalty is absolutely out of the question (or tracing exception handlers is a must), then turn this option ON before starting the target application, run target application until vectors have been changed (until breakpoint or manual stop) and then turn this option OFF and continue debugging.

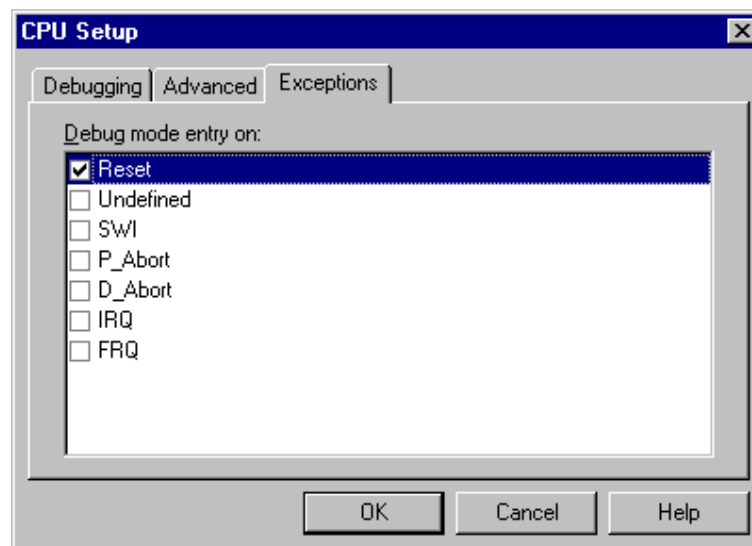
**IMPORTANT NOTE:** In order for this option to function correctly, only certain types of instructions can be used as exception vectors. A direct branch (“b”) and PC load (“ldr pc,[pc...]”) instructions are supported. If any other type of instruction is used as exception vector, the debugger will STOP the target CPU execution as soon as that exception vector is invoked.

### *Use Handshaking*

When this option is checked, execution of every command is handshaked. This is required by CPUs for which the JTAG clock is too high.

## **3.4 Exceptions**

XScale CPUs are ARM core based and so have the same set of exceptions as ARM CPUs.



*CPU setup, Exceptions menu*

The debug mode will be entered (that is, the CPU will stop) when any of the selected exceptions occurs during program execution.

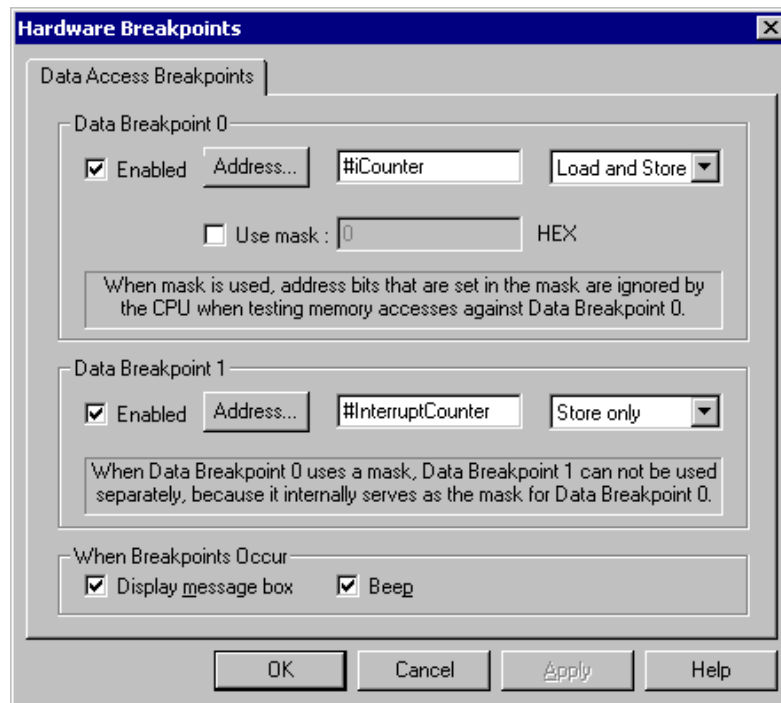
## 4 Real-Time Memory Access

winIDEA debugger provides Real-time access to target memory. This means that variable values and memory contents can be observed while the CPU is running. Real-time access incurs a penalty of 11 instruction cycles for each 32-bit word read while target application is running. Write is also possible, but incurs a much higher penalty.

Watch window's *Rt. Watch* panes can be configured to inspect memory without stalling the CPU. Optionally, memory and SFR windows can be configured to use real-time access as well.

Please refer to the Software User's Guide for more information on Real-Time watches.

## 5 Access Breakpoints

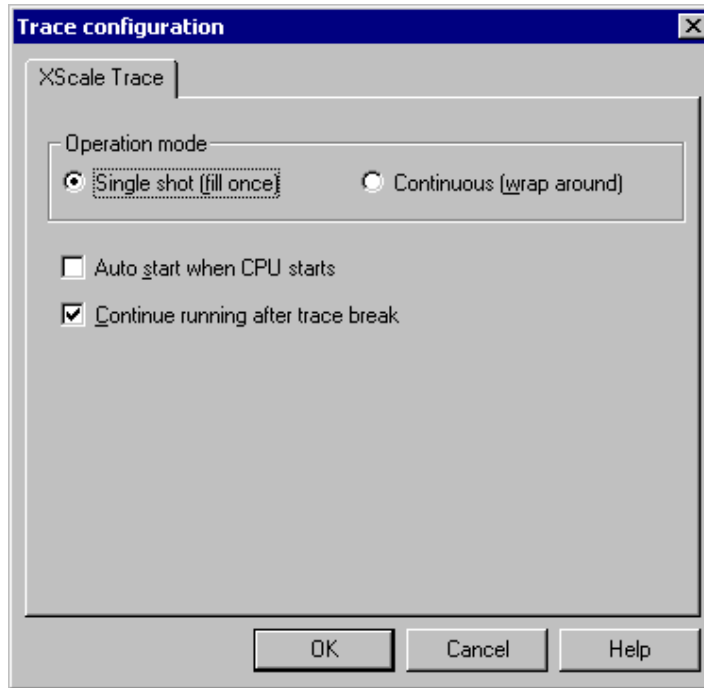


*XScale Hardware Breakpoints page*

XScale debug hardware offers two data access breakpoints. They can either be used independently to trap data accesses to two different memory locations or they can be used in a combination where the second data breakpoint (DB1) provides the address mask for the first data breakpoint (DB0), which allows for data accesses to be trapped across one or more memory ranges, depending on the mask. Bits that are set in the mask are ignored by the processor when comparing the address of a memory access with the address in DB0.

A data breakpoint can be configured to trap load only, store only or both load and store accesses to the specified location or range(s). The address of the location can be given as a numeric address in hexadecimal or as a symbol chosen using the "Address..." button on the dialog.

## 6 Trace Configuration



*XScale Trace Configuration*

XScale debug hardware includes an on-chip trace buffer that is 256 bytes long and can record the history of program execution (there is no bus cycle or data trace). Trace hardware records history in meta data format which is analyzed by host debugger and usually results in a history of instructions whose count will often be a lot higher than just 256 bytes. Trace buffer can be operated in one of two modes, “single shot” mode or “continuous” mode.

In “single shot” mode, trace buffer starts recording as soon as target CPU goes into running. If no other debug exceptions occur before trace buffer fills up, then the CPU will stop when trace buffer fills up. CPU can also come to a stop for other reasons before trace buffer fills up (breakpoint, exception trapped or user break). In such a case there will simply be less trace data and therefore less history to display. If checkbox “Continue running after trace break” is checked and the CPU stops because trace buffer filled up, then recorded data will be retrieved from the on-chip buffer and the CPU will be put right back into running, but without starting the trace recording.

In “continuous” mode, trace buffer also starts recording as soon as target CPU goes into running. However, CPU does not stop when the buffer fills up in “continuous” mode. Rather, the trace buffer keeps recording in a FIFO manner until the CPU comes to a stop for whatever other reason such as hitting a breakpoint, exception being trapped or user break. So when CPU does come to a stop when trace is recording in “continuous” mode, trace buffer will contain the most recent trace entries up to and including the last instruction that was executed before CPU stopped.

If checkbox “Auto start when CPU starts” is checked, trace will automatically start recording every time CPU is put from a non-running state into a running state.

### ***Restrictions due to on-chip debug module***

Trace may stop recording prematurely in continuous mode when real-time memory access is used.

## 7 Getting Started

Depending on the target system being used, a few specific things might need to be taken care of for successful operation.

### *Notes when using PXA250 processor*

If a target system with PXA250 processor does not have any bootable code that would perform target system initialization, then some initialization would have to be performed through the use of Hardware->Emulation options->Initialization dialog in order to configure the on-board memory controller for access to target's system memory. This would make sure that target's system RAM is accessible before an attempt is made to download code into target memory.

### *Cogent CSB226 notes*

CSB226 is equipped with XScale PXA250 processor. JTAG scanning speed may need to be lowered to "Medium" in Hardware->Emulation options->JTAG in order to ensure reliable operation on CSB226. Use of higher JTAG scan speed is often dictated by the quality of the signal traces between the emulator and processor's JTAG pins.

## Debug Connection

Normally, the minimum settings required by the emulator to be able to connect to the target CPU are specifying the emulator type, the communication type, the CPU type, specifying the required JTAG speed and specifying Debug I/O Levels.

- Next, verify if the JTAG connector in the target matches with the pinout defined by the CPU vendor. The required connector pinout can be also found in the hardware reference document delivered beside the debug iCARD.
- Connect the emulator to the target.
- First power on the emulator and then the target! When switching off the system, switch off the target before the emulator!
- Close all debug windows in winIDEA except for the disassembly window.
- Execute debug CPU Reset command.

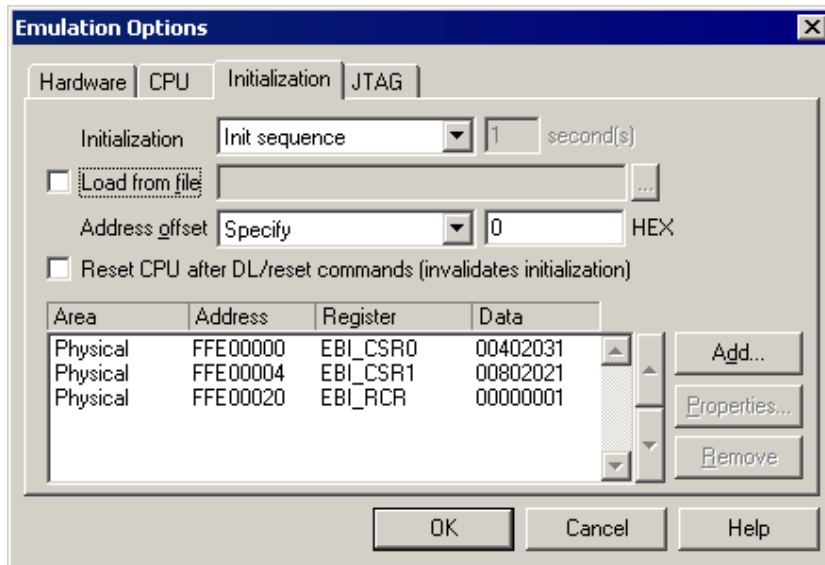
WinIDEA should display STOP status and disassembly window should display the code around the address where the program counter points to.

Next step is to download the code in the RAM memory or program the code in the flash.

Before the flash programming or download can take place, the user must ensure that the memory is accessible. This is very important since there are many applications using memory resources (e.g. external RAM, external flash), which are not accessible after the CPU reset. In that case, the debugger must execute after the CPU reset a so called initialization sequence, which configures necessary CPU chip selects and then the download or flash programming can actually take place. The user must set up the initialization sequence based on his application.

The initialization sequence can be set up in two ways:

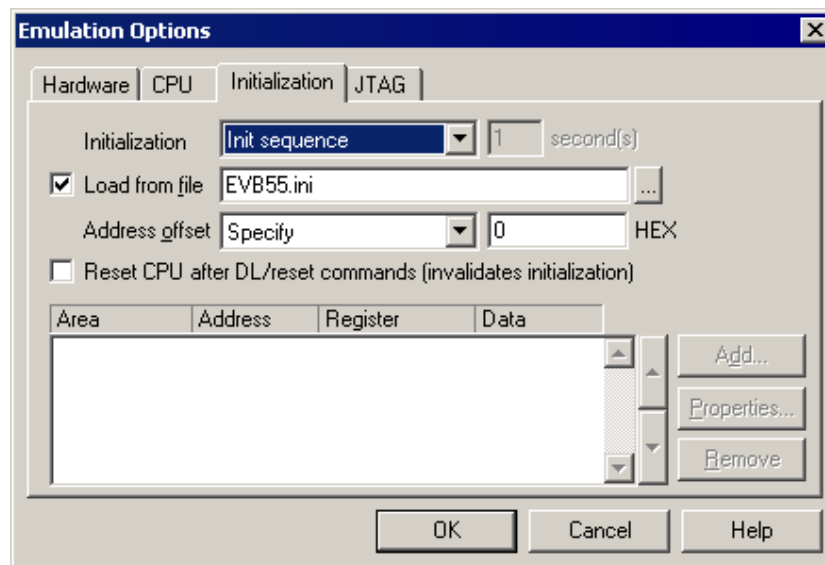
1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



- winIDEA accepts initialization sequence as a text file with .ini extension. The file must be written according to the syntax specified in the appendix in the hardware user's guide.

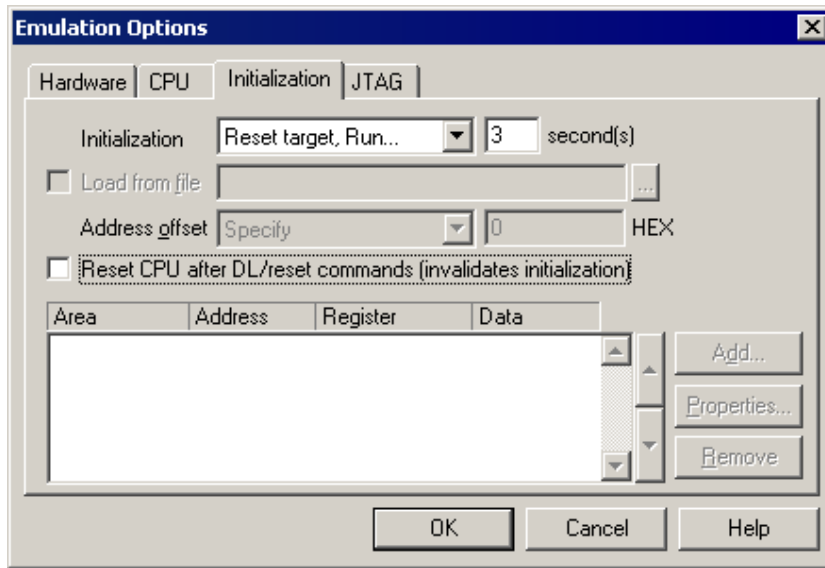
Excerpt from EVB55.ini file for the Atmel AT91M55800 CPU (ARM7TDMI):

```
S EBI_CSR0 L 0x00402031 // CS0 - ext. flash, 4 wait states
S EBI_CSR1 L 0x00802021 // CS1 - ext. SRAM
S EBI_RCR L 0x00000001 // remap internal RAM
```



The advantage of the second method is that you can simply distribute your .ini file among different workspaces and users. Additionally, you can easily comment out some line while debugging the initialization sequence itself.

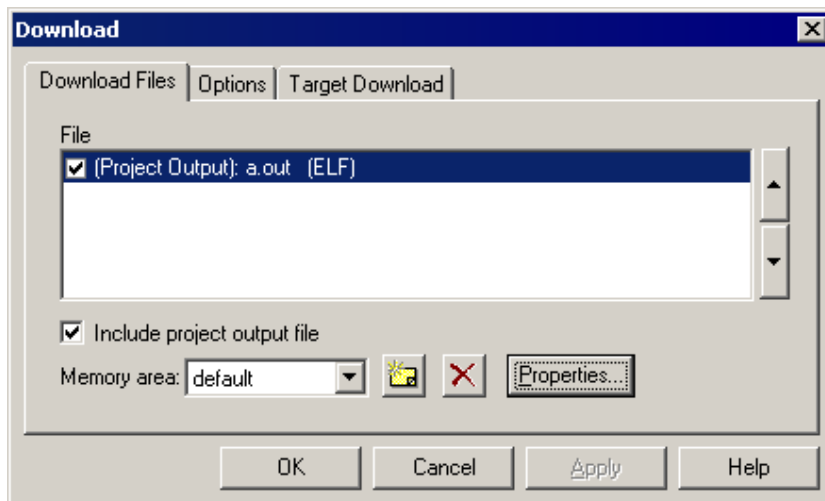
There is also a third method, which can be used too but it's not highly recommended for the start up. The user can initialize the CPU by executing part of the code in the target ROM for X seconds by using 'Reset and run for X sec' option.



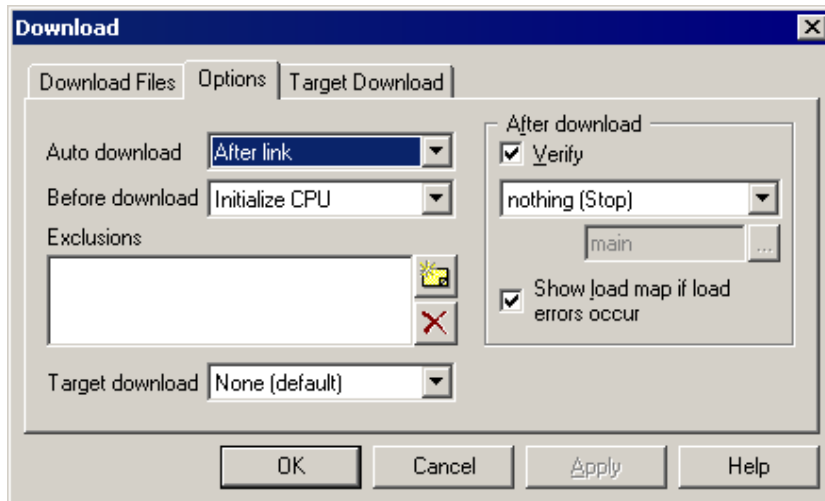
## Debug Download

Debug download is used to load the debug info to the debugger, to load the code in the target RAM memory and on some CPUs also to load the code in the internal flash. Otherwise, normally 'FLASH Programming Setup' dialog needs to be invoked to program the flash.

- Specify file(s) to be downloaded in the 'Debug/Files for download/Download Files' tab.

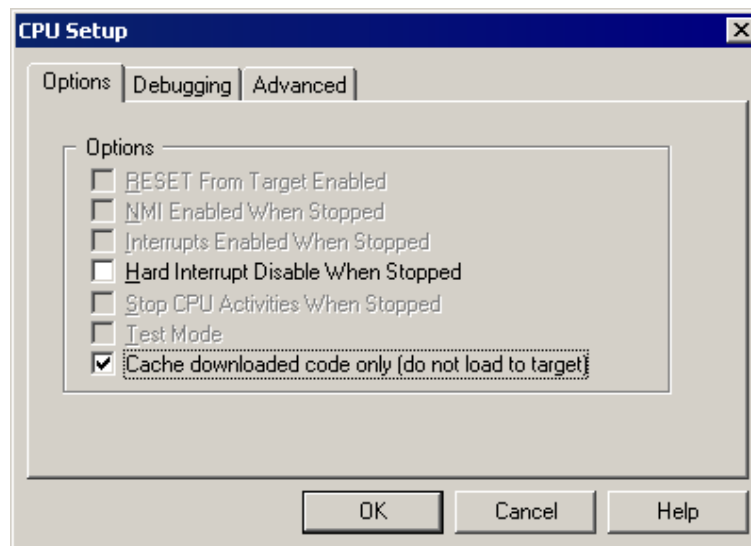


- Make sure that 'Initialize CPU' before download is configured in the 'Options' tab. This yields in the initialization sequence (explained earlier) being executed before the actual download.



- It's recommended to check the 'Verify' option in the 'Options' tab. Then WinIDEA pops up a warning in case of download error(s).

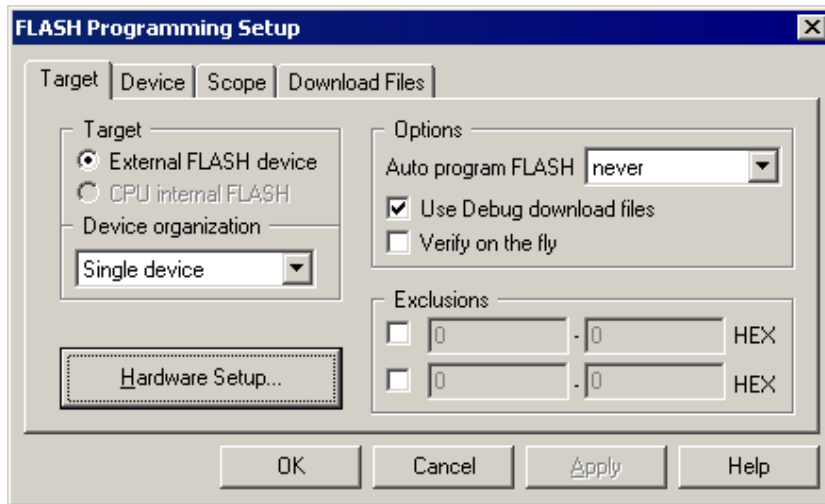
When debugging the application with the code in the target ROM, it is only necessary to download the debug info for that memory area. Code can be excluded (click on the file and press 'Properties' button). However, if the file specified here, is used for the flash programming too, then keep the code included and check 'Cache downloaded code only (do not load to target)' option. When this option is checked, memory writes don't propagate to the target during debug download. There is also no need for that if there is a flash in the target, which requires special programming algorithm. In worst case, debug session may even misbehave if memory writes propagate to the target flash memory during debug download.



## Flash Programming

To program the external flash device you need to invoke the 'FLASH Programming Setup' dialog (FLASH/Setup...). Let's program AMD 29LV160DB flash located at 0xFFE00000 address as an example.

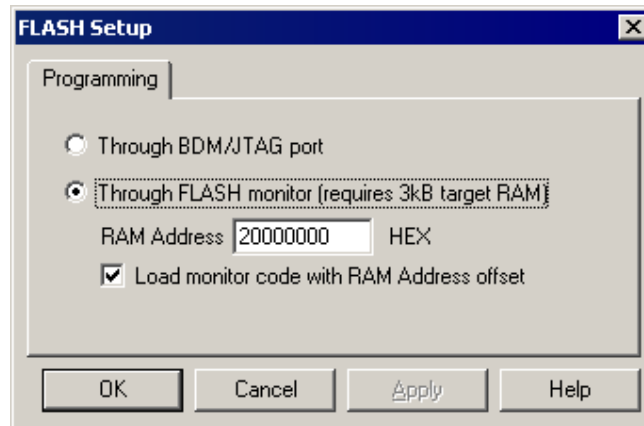
- First, it's necessary to select ('Target' field) whether the CPU's internal or external flash is programmed.



When the CPU internal flash is programmed, winIDEA takes care of most of the necessary settings. For the external flash, flash device has to be selected and start address set when programming the external flash device.

Note that winIDEA uses 'external flash programming' mechanism also for some internal CPU flashes (Philips LPC2000 family, Atmel AT91FR4081, AT 91RV40162 ...)

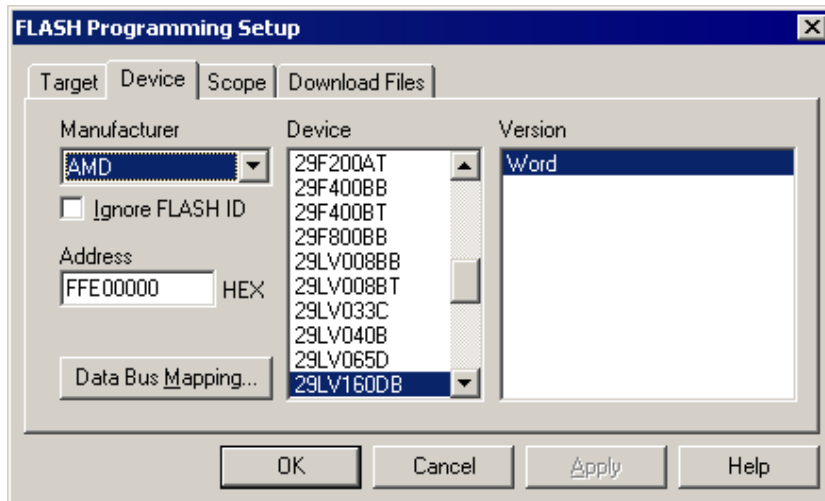
- Next, select flash programming type. WinIDEA supports flash programming through the debug JTAG port and fast FLASH monitor. Press 'Hardware Setup...' button in the 'Target' tab in the 'FLASH Programming Setup' dialog for the selection.



Normally, the user should go straight for fast FLASH monitor use. Programming through the JTAG port is much slower and recommended to be used when troubleshooting flash programming. Flash programming through FLASH monitor requires up to 3kB of target RAM, where flash programming monitor is loaded and then the flash programming algorithm executed. The user needs to enter the target RAM address and make sure that the target RAM is accessible before flash programming starts. Use the initialisation sequence to enable access to the RAM if it's not accessible after the CPU reset.

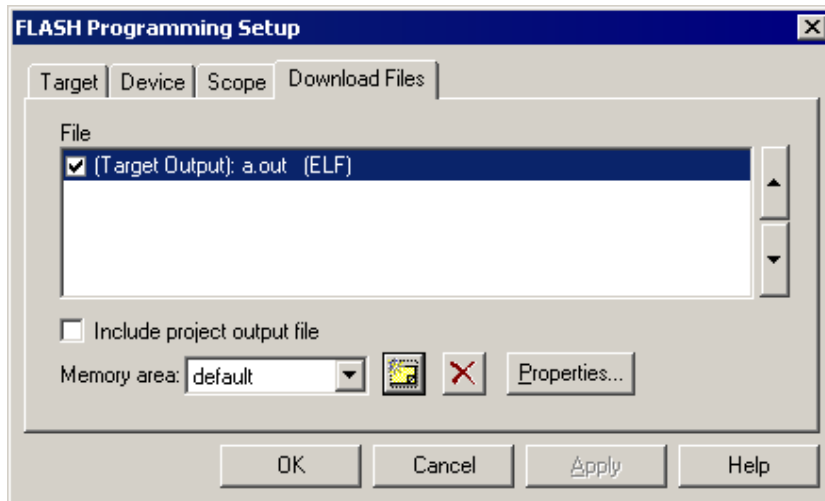
Flash programming through JTAG port is not supported for some flashes where custom FLASH monitors are written. WinIDEA pops up a warning when programming through JTAG port is not supported,

- Next, define the device to be programmed and its start address.

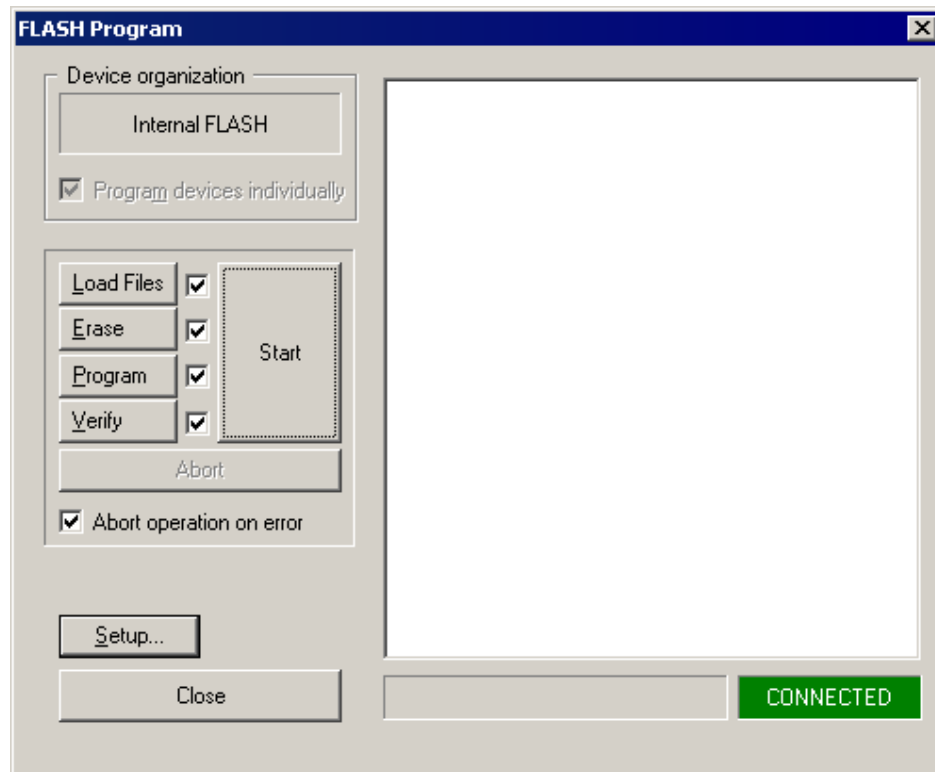


- Finally, the file to be programmed needs to be defined. It can be added in the 'Download files' tab within the 'FLASH Programming Setup' dialog.

The alternative is to specify file(s) in the 'Debug/Files for Download/Download files' tab, where normally files for debug Download are specified. Make sure that 'Use Debug download files' option ('Target' tab in the 'FLASH Programming Setup' dialog) is checked then. In first case, the option must be unchecked.

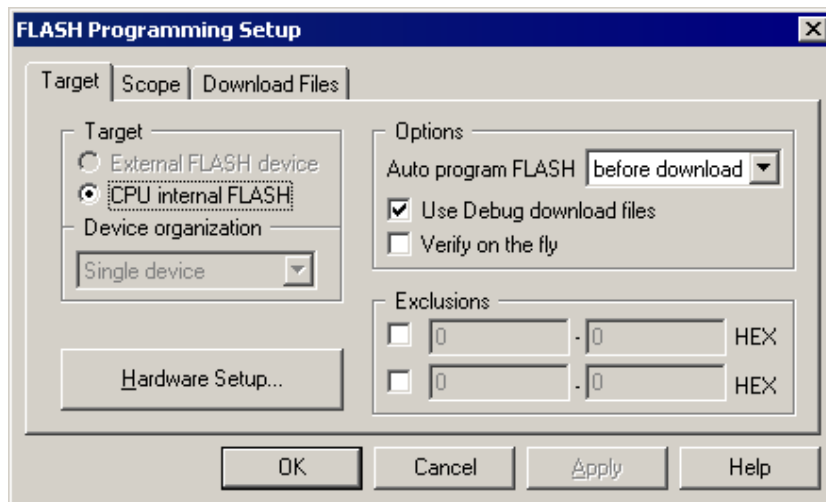


- 'FLASH Program' dialog should be invoked from the 'FLASH' menu after the flash programming is configured.



Check boxes beside Load Files, Erase, Program and Verify buttons should be checked and flash programming started by pressing 'Start' button. During the flash programming, a status and eventual errors are displayed in the dialog.

The debugger can program the flash automatically before the download. Then 'before download' in the 'Auto program FLASH' combo box must be selected.



Refer to hardware user's guide for more details on flash programming.

The debugger should be now operational assuming that the code is loaded in the target RAM or programmed in the target flash and the debug info loaded to the debugger. The user should be able to reset, run, stop the application, carry out instruction and source single step, set BPs, etc.

## 8 Troubleshooting

- When performing any kind of checksum, remove all software breakpoints since they may impact the checksum result.
- Make sure that the power supply is applied to the target JTAG connector when 'Target VCC' is selected for Debug I/O levels in the Hardware/Emulator Options/Hardware tab, otherwise emulation fails or may behave unpredictably.

## 9 Notes

- There are a couple of important limitations that must be considered by target software implementors when using any debugger for XScale platform. The code running on an XScale target must never use the coprocessor instruction MCR which invalidates a single I-cache line to perform the invalidate operation on the exception vector table. Also, target code must never perform I-cache line invalidate operation in the range where debug handler is located, which is a 2KB range starting from debug handler base address. Failure to observe these limitations would cause the debug session to fail after the offending Icache instruction is executed, because debugger would loose the control of the target. However, the coprocessor MCR instruction for "global invalidate I-cache" does not affect the debugger.

---

Disclaimer: iSYSTEM 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 herein.

© iSYSTEM. All rights reserved.