Intel® C++ Software Development Tool Suite - v2.0.1
For Intel XScale® Microarchitecture, Professional

Release Notes

Document Number: 280124-023US

Contents

Overview

The *Intel® C++ Software Development Tool Suite - v2.0.1 For Intel XScale® Microarchitecture, Professional* (Intel® SDT 2.0.1) is a sophisticated tool set designed for software development for PDAs, handheld devices, and cellular phone solutions. Developed to fully utilize applications designed for Intel® Personal Internet Client Architecture (Intel® PCA) processors based on Intel XScale® technology, the Intel® SDT 2.0.1 contains a compiler system as well as a set of high level language debuggers.

The Intel® C++ Compiler offers high multimedia performance for new PDA and wireless applications. It includes support for Intel® Wireless MMX™ 2 technology (Intrinsic functions, Vectorizer support). The compiler and debuggers are compatible with Palm OS* v5.x and Nucleus* OS v1.14, and can also be used for OS independent software development. Additionally, debugger support is included for Symbian OS* v8 EABI and EKA2 and v9 as well as for Linux* Kernels 2.4.x/2.6.

Optionally you can install the GPL kernel driver and patches for the Linux* Application Debugger which permit access to the processor traces, the hardware breakpoints and the Intel® Wireless MMX™ 2 technology registers. Install this plug-in separately after installing Intel® SDT 2.0.1.

One other optional installation is a Plug-in to the Metrowerks* CodeWarrior* IDE version 4.x and 5.x, as an alternative to the Intel® Integrated Development Environment (Intel® IDE). Install this plug-in separately after installing the Intel® SDT 2.0.1.

Target processor support includes the Intel® PXA25x, Intel® PXA26x and Intel® PXA27x applications processor families as well as the Intel® PXA800F and the Intel® PXA9xx cellular processors family (Intel XScale® microarchitecture part only).

Please refer also to the

- Installation Guide (`<install_dir>\Doc\INSTALL.htm`) for product installation,
- README (`<install_dir>\Doc\README.txt`) for information about installation directory structure, Windows* Start menu entries and additional information,

Note: `<install_dir>` is the installation path of the tool suite, default: `C:\Program Files\Intel\SDT2.0.1`. The name of the path, `C:\Program Files\`, may be different depending on your Windows* operating system.

Product Support

This product release provides support for target OS, target CPU, JTAG interfaces and flash types as follows:

Support	Description	Version
Target OS	Symbian OS*	v8.1/v9
	Palm OS*	v5.x
	Nucleus OS*	v1.14
	Linux* (Debugger only)	Kernel 2.4/2.6
	OS independent development	NA
Target CPU	Intel® PXA25x Applications Processor Family	NA
	Intel® PXA26x Applications Processor Family	NA
	Intel® PXA27x Applications Processor Family	NA
	Intel® PXA800F Communications Processor (Intel XScale® microarchitecture part only)	NA

	Intel® PXA9xx Communications Processor Family (Intel XScale® microarchitecture part only)	NA
Target Interfaces	Intel® JTAG Cable	rev. 1 and 2
	Macraigor OCDemon* Raven ARM* 20 JTAG interface	NA
Flash Types	Intel StrataFlash® Wireless Memory System LV18/LV30	NA
	Intel StrataFlash® Wireless Memory L18/ L30	NA
	Intel® Wireless Flash Memory W18/W30	NA
	Intel StrataFlash® Flash Memory J3D	NA
	Intel StrataFlash® Flash Memory K3/K18 *(discontinued products)*	NA
	Intel StrataFlash® Embedded Memory P30	NA
	Intel® Advanced+ Boot Block Flash Memory type B and C	NA
	Intel® Flash Memory D18/D30, E18/E30	NA
	All processor-internal flashes on Intel® PXA26x/27x/800F/9xx	NA

Product Contents

This product release contains the components as follows:

Contents	Description	Version
Intel® Compiler Tools	Intel® C++ Compiler	2.0.26
	Intel® C/C++ Libraries	2.0.43
	Intel® Library Manager	1.3.16
	Intel® Assembler	2.2.58
	Intel® Linker	2.0.61
Intel® Object Converters	Intel® DWARF2BD	2.9.24
	Intel® ELF2BIN	1.1.7
Intel® XDB Debuggers	Intel® XDB JTAG Debugger	2.8.14
	Intel® XDB Simulator Debugger	2.8.15

	Intel® XDB Application Debugger for Linux* (*also referred to as* Intel® XDB ROM Monitor Debugger)	2.8.13
Intel® XDB OS Awareness Plug-ins	Intel® XDB OS Awareness Plug-in for Palm OS*	2.2.1.0
	Intel® XDB OS Awareness Plug-in for Symbian OS*	2.5.2.0
	Intel® XDB OS Awareness Plug-in for Nucleus* OS	2.7.1
	Intel® XDB OS Awareness Plug-in for Linux* System Debugging	2.2.2
	Intel® XDB OS Awareness Plug-in for Linux* Application Debugging	1.1.1.0
Intel® XFlash	Intel® JTAG Flash Memory Programmer	1.5.4
Intel® IDE	Intel® Integrated Development Environment	1.2.17
Intel® CodeWarrior* Plug-In	Optional plug-in for Metrowerks* CodeWarrior* Integrated Development Environment.	1.4.1.0
Product Documentation	The product documentation is provided for easy access of all the documents. It's located at `<installdir>\doc\doc_index.htm`	NA

System Requirements

Host System

- A computer based on an Intel® Pentium® processor or subsequent IA-32 based processor. (Intel® Pentium® 4 processor recommended).
- 128 MB (256MB recommended).
- 100 MB of disk space, plus an additional 200 MB during installation for the download and temporary files.

Host Software

- Windows* 2000 Professional or Windows* XP Professional
- Adobe* Acrobat Reader* version 4.0 or later is required to view some of the product documentation
- A Web browser is required to view some of the HTML-based product documentation
- (Optional) Metrowerks* CodeWarrior* IDE version 4.x or 5.x, if Intel® CodeWarrior* Plug-in is used.

Target Interface (using JTAG)

When using the JTAG interface for debugging, one of the following JTAG interfaces is required and must be properly installed with Parallel Port Settings in the System BIOS:

Interface	Parallel Port Mode	Parallel Port Channels

Intel® JTAG Cable	ECP	I/O 0x378, IRQ7, DMA0 (recommended settings, but any other channels may work)
Macraigor OCDemon* Raven ARM* 20 JTAG interface	EPP (recommended) or ECP	I/O 0x378, IRQ7, DMA0 (recommended settings, but any other channels may work)

Notes:

- Some PCs need an extra reboot after changing the parallel port setup in the CMOS BIOS.
- Older Macraigor OCDemon Raven ARM* 20 JTAG devices (short flat ribbon cable with 20-pin JTAG header, but without daughter-box) may not work with Intel® PXA900 DVK/PDK boards due to insufficient voltage strength on VTref while operation.

Compatibility

ARM* ABI/EABI Compliance

By default, the Intel® SDT 2.0.1 supports ARM* EABI v2.0 whereas Intel® SDT 2.0 used ARM* EABI v1.0 and the predecessor version Intel® SDT 1.2 used ARM* ABI by default. A compiler switch allows the Intel® SDT 2.0.1 to generate backwards-compatible ARM* ABI, EABI v1.0 or EABI v2.0 code.

IMPORTANT NOTE on using C++ Exception Handing:
All projects using C++ Exception Handling, must be re-built with the Intel® SDT 2.0.1 release version since the EABI C++ exception handling model used in SDT 2.0.1 is different from the model used in Intel® SDT 2.0 or Intel® SDT 1.2.

Installation

Please see the separate Installation Guide `INSTALL.htm` for installation and configuration instructions. It's located at the root directory of the installation CD or the extracted `w_xisdt_p*.exe` file.

What's New with Intel® SDT 2.0.1

The following section discusses new features and changes coming with Intel® SDT 2.0.1 Update Release:

ARM* EABI v2.0 Support

Note: Please refer to What was New with Intel® SDT 2.0 for EABI v1.0 support provided with the Intel® SDT 2.0 release.

The Intel® 2.0.1 Update Release now supports the ARM* Embedded Applications Binary Interface (EABI) v2.0 which is the default setting. The existing compiler/assembler/linker option `-abi=<version>` is now extended as follows:

Compiler/Assembler/Linker EABI Option Extension

Option	Parameter	Description
`-abi`	`EABI2`	**Default value.** Support for ARM* Embedded Applications Binary Interface (EABI) v2.0
	`EABI1`	Support for ARM* Embedded Applications Binary Interface (EABI) v1.0 *Choose this parameter for backwards compatibility to Intel® SDT 2.0 only!*
	`ATPCS`	Support for the ARM*/Thumb* Procedure Call Standard with extensions to support Intel® Wireless MMX™ technology. *Choose this parameter for backwards compatibility to Intel® SDT 1.2 only!*

New Libraries

To support ARM* EABI v2.0, a new set of libraries was introduced as follows:

```
x0__ac50.a
x0__ax50.a
x0__as50.a
x0__a050.o
x0__ac53.a
x0__ax53.a
x0__as53.a
x0__a053.o
```

The Intel® Library names follow our standard naming conventions (refer to Documentation). The two additionally set bits encode EABI v2.0 compliance and an EABI compliant change in the memory layout of double floating point values. For example, the ATPCS absolute C standard library is called `x0__ac00.a` whereas the EABI absolute C standard library is called `x0__ac50.a`.

Notes:

- Rebuilding of older legacy projects requires **adopting** of the selected libraries included in the link stage, otherwise the build won't be successful.
- New projects should use EABI v2.0 from the start. Old projects should be recompiled to use EABI v2.0, if possible.
- Libraries are selected automatically based on the command line options, if the linker is implicitly called by the call of the compiler.

Optimization Support for Intel® PXA9xx Processors Family

The Intel® C++ Compiler uses the existing option `/QTPxsc2` to enable support for the Intel® PXA9xx processors family.
The Intel® Assembler uses the existing option `/mcpu 2` to enable support for the Intel® PXA9xx processors family.

Note: Support for Intel® PXA9xx is limited to the Intel XScale® microarchitecture core.

Debug Support for Intel® PXA9xx Processors Family

The Intel® XDB JTAG Debugger and Intel® XDB Simulator recognize the Intel® PXA9xx processors family and can display its full register set.

Note: Debug visibility is limited to the Intel XScale® microarchitecture core of the Intel® PXA9xx processors family.

New startup and memory initialization scripts coming with the Intel® XDB JTAG Debugger and Intel® XDB Simulator:

Debugger Scripts	Description
`pxa900.xdb`	Initialization scripts (SRAM, GPIO) for Intel® PXA9xx
`pxa900_32bit.xdb`	Initialization scripts (external SRAM, GPIO) for Intel® PXA9xx SDK/DVK development boards.
`pxa900.xsf`	Debugger / Simulator startup file for Intel® PXA9xx

The Intel® XDB Flashwriter from the Intel® XDB Browser for JTAG now supports flashes for Intel® PXA9xx based targets as follows:

Flash Configuration Files	Description
`pxa900_32bit.fcf`	For burning external flashes (32-bit memory access) on flashbase=0x04000000 (used for Intel® PXA9xx SDK/PDK development boards).
`pxa900_ext.fcf`	For burning external flashes on flashbase=0x04000000
`pxa900_cpu.fcf`	For burning internal Intel XScale® microrchitecture flash on flashbase=0x80000000
`pxa900_dsp.fcf`	For burning internal Intel® MSA flash on flashbase=0x84000000

You can run also the following flash batch scripts `*.xdb` in the Intel® XDB JTAG Debugger to flash memory:

Debugger Flash Batch Scripts	Description
`pxa900_32bit_burn.xdb`	For burning external flashes (32-bit memory access) on flashbase=0x04000000 (used for Intel® PXA9xx SDK/PDK development boards).
`pxa900_ext_burn.xdb`	For burning external flashes on flashbase=0x04000000
`pxa900_cpu_burn.xdb`	For burning internal Intel XScale® microrchitecture flash on flashbase=0x80000000
`pxa900_dsp_burn.xdb`	For burning internal Intel® MSA flash on flashbase=0x84000000

Intel® XFlash Support for Intel® PXA9xx

The Intel® JTAG Flash Memory Programmer (Intel® XFlash) supports new options for the Intel® PXA9xx processors family as follows:

Option	Parameter	Description
`-platform`	`PXA900_32bit`	To flash the external flash memory (32-bit access) of an Intel® PXA9xx SDK/PDK.
	`PXA900_EXT`	To flash external flash memory of other DVK/EVB boards based on Intel® PXA9xx.

	PXA900_DSP	To flash the internal flash of the Intel® MSA core of the Intel® PXA9xx processor
	PXA900_CPU	To flash the internal flash of the Intel XScale® core of the Intel® PXA9xx processor
-flashname	B|C|J3|K3|L18|K18| LV18|LV30|W30|P18|W30|	Flash types supported by Intel® XFlash

What was New with Intel® SDT 2.0

The following section discusses new features and changes that were released with Intel® SDT 2.0.

ARM* EABI v1.0 Support

A new option `-abi=[eabi|atpcs]` enables/disables ARM* Embedded Applications Binary Interface (EABI) v1.0 compliance. This option has to be used with the same arguments in all code generating tools (Intel® Compiler, Intel® Assembler, Intel® Linker). By default, the `eabi` parameter is set for Compiler, Assembler and Linker to produce and link code in EABI compliant mode. Parameter `atpcs` for Compiler, Assembler and Linker creates and links backwards compatible ARM* Application Binary Interface (ABI) code as it was generated with the Intel® SDT 1.2.
Examples:
The following commands will produce a backward compatible object file and executable file, respectively.

```
    ccxsc -abi=atpcs -c foo.c
    ccxsc -abi=atpcs foo.o
```

The Intel® Assembler and Intel® Linker calls are used in the same way.
Examples:

```
    asxsc foo.s
    asxsc -abi=atpcs foo.s
    ldxsc ob1.obj ob2.obj
    ldxsc -abi=atpcs obj1.obj ob2.obj
```

Note: This option has to be used consistently for all code generating tools.

Intel® SDT 2.0.1 does not support all ELF features. Linking objects from third-party tools might fail in rare cases.

Supported ABIs

The Intel® SDT 2.0 supports compliance to the following ABIs:

- ATPCS: Compatible to ARM* ADS 1.2
- EABI: Compliance to ARM* EABI v1.0

Note: ATPCS may not be supported in future releases. We recommend to use EABI for new projects.

New ABI-related Components

The Intel® SDT 2.0 will have the following ABI-related new components:

A new set of libraries to be used for EABI projects:

```
    x0__ac30.a
    x0__ax30.a
    x0__as30.a
    x0__a030.o
    x0__ac33.a
    x0__ax33.a
    x0__as33.a
    x0__a033.o
```

The Intel® Library names follow our standard naming conventions (refer to Documentation). The two additionally set bits encode EABI v1.0 compliance and an EABI compliant change in the memory layout of double floating point values. For example, the ATPCS absolute C standard library is called `x0__ac00.a` whereas the EABI absolute C standard library is called `x0__ac30.a` and so on.
Note: Rebuilding of older legacy projects requires renaming of the libraries included in the link stage, otherwise the build won't be successful.

ARM* ABI Backwards Compatibility

Intel® SDT 2.0 used with setting `-abi=atpcs` is generally backwards compatible to Intel® SDT 1.2 with a few exceptions: The EABI C++ exception handling model is different from the exception handling model used in Intel® SDT 1.2.

Intel® SDT 2.0 used with default setting `-abi=eabi` is not compatible to Intel® SDT 1.2.

Porting Intel® SDT 1.2 Projects

The Intel® SDT 1.2 supported only ATPCS. The Intel® SDT 2.0 supports ATPCS and EABI, whereby it defaults to EABI. In order to port SDT 1.2 projects to SDT 2.0, one has basically two options: Port the project to EABI (recommended) or use SDT 2.0 in ATPCS mode.

- Porting to EABI: This requires that the entire project is available in source code and that the entire project is ported. Linking to SDT 1.2 compiled binaries may result in run-time errors. In order to port the project, the project only needs to be recompiled. Assembly source files need to be ported to EABI manually. If you specify libraries directly, you need to change the library names to the EABI variant. We recommend to port existing projects to EABI.
- Using SDT 2.0 in ATPCS mode: This requires that all sources that use C++ exception handling are available in source code. Linking to SDT 1.2 compiled binaries not using C++ exception handling is possible. In order to port the project, the option `-abi=atpcs` has to be added to all compiler, assembler, and linker calls. Sources that use C++ exception handling need to be recompiled; the project needs to be re-linked. We recommend recompiling the entire project. This option is required for projects that need to be compatible to ARM* ADS 1.2 or to Intel SDT 1.2.

New Intel® C++ Compiler Command Line Options

The following new command line options are accepted by the Intel® C++ Compiler with the semantics as described in the following table:

Option	Description
`-abi=`*`callstandard`*	Specifies the procedure call standard, depending on the `callstandard` parameter. `-abi=eabi` specifies ARM* C/C++ Embedded Applications Binary Interface standard (EABI). This is the default setting. `-abi=atpcs` specifies ARM*/Thumb* Procedure Call Standard (ATPCS)
`-fint-enums`	The option causes the compiler to switch on int enum-generation. This is default if `-abi=atpcs` is set.
`-fshort-enums`	The option causes the compiler to switch on short enum-generation. This is default if `-abi=eabi` is set.
`-fshort-wchar`	The option causes the compiler to treat wchar_t as unsigned short. This is default if `-abi=atpcs` is set.
`-fuint-wchar`	The option causes the compiler to treat wchar_t as unsigned int. This is default if `-abi=eabi` is set.
`-fsigned-char`	The option changes the default char type to signed type. This is default if `-abi=atpcs` is set.
`-funsigned-char`	The option changes the default char type to unsigned type (same as `-J`). This is default if `-abi=eabi` is set.
`-MXA`	The option causes the compiler to generate file dependencies for use with the Workbench excluding system headers. Apart from excluding the system headers, this option is identical to the `–MX` command line option.
`-platform=symbian`	Allows generation of Symbian OS* DLLs and executables.
`-QRthumb`	This option switches from ARM*-code (default) to THUMB code generation used with the ARM* mode compiler ccxsc. Calling the ARM* mode compiler `ccxsc -Qrthumb` is an equivalent to calling the Thumb mode compiler `ccxscthb.`
`-Qvariable_locations-`	Improves compatibility of the debug information when interpreted by GDB
`-Zi-`	This is a modification of the existing `–Zi` option. `-Zi-` disables the generation of symbolic debug information. The generation is disabled as default.

New Intel® Linker Command Line Option

The following new command line option is accepted by the Intel® Linker with the semantics as described in the following table:

Option	Description
`-platform=symbian`	Allows generation of Symbian DLLs and executables.
`-robase=`*`address`*	Replaces previous option -base. Define the RO base address when using default layout
`-rwbase=`*`address`*	Define the RW base address when using default layout

Support for Intel® Wireless MMX™ 2 Technology

The compiler supports the Intel® Wireless MMX™ 2 technology which are the Intel® New Media Technology Extensions to the existing Intel® Wireless MMX™ technology:

- The Intel® C++ Compiler uses the new option `-QTPxsc4` to enable the use of Intel® Wireless MMX™ 2 technology via intrinsic functions.
  Note: Intel® Wireless MMX™ 2 technology support will not be extended to the vectorizer as there are no instructions that can benefit from vectorization. There are no changes needed in the Intel® Linker to obtain Intel® Wireless MMX™ 2 technology support.
- The Intel® Assembler uses the new option `-mcpu 4` for using Intel® Wireless MMX™ 2 technology intrinsics.
- The Intel® XDB Simulator Debugger provides a new option `PXA27x_Wireless_MMX2` in the startup dialog to support Intel® Wireless MMX™ 2 technology instructions.
  Note: The implementation provides functional simulation only. Cycle accuracy is not supported.

PC Relative Addressing in Assembler with Position-independent Code

PC relative addressing in the Assembler for Position-independent Code (PIC) has been improved. Please refer also to PC relative addressing in PIC code issues with older Assembler versions.

Additional Mnemonics for Intel® Wireless MMX™ 2 Configuration/Status Registers

The Assembler now also accepts Intel® Wireless MMX™ 2 Configuration/Status Register names in the format of `wC<`*x*`>,` where `<`*x*`>` represents the register number of Coprocessor 1. The mnemonics which can be used synonymously are shown in the following table:

Register Name	Coprocessor Register Number	Alternative Mnemonics	
Coprocessor ID, Revision, Status	0	wCID	wC0
Control	1	wCON	wC1
Saturation SIMD flags	2	wCSSF	wC2
Arithmetic SIMD Flags	3	wCASF	wC3
Reserved	4..7	-	-
General Purpose Register 0	8	wCGR0	wC8
General Purpose Register 1	9	wCGR1	wC9
General Purpose Register 2	10	wCGR2	wC10

General Purpose Register 3	11	wCGR3	wC11
Reserved	12..15	-	-

Automatic Thumb* Code Detection

The Intel® XDB Debugger trace detects Thumb* code automatically. Manually defining the Thumb* mode areas is not needed any more.

36-Bit Page Translation Table Support

A new page translation table dialog is available in the Intel® XDB Debugger to configure 36-bit page translations. It allows to map 32-bit virtual to the 36-bit physical addresses.

Flash Simulation Plug-in

A new Intel® XDB Flash Simulation Debugger Plug-in simulates flash memory to enable development of flash access in a simulated environment.

The plug-in currently supports simulation of Synchronous Intel® StrataFlash memory, 28FxxxKx, for example 28F128K3, 28F256K3, 28F640K3.

The Flash simulation Plug-in is able to simulate a parallel combination of two flashes, which means a usage of two 16-bit flashes as one 32-bit flash with doubled size. The following custom types can be chosen, for example 2x28F128K3, 2x28F256K3, 2x28F640K3.

Another feature is the possibility to allocate several flashes at the same time in the address space, to allow flash memory constellations like in real hardware.
The simulated flash can be controlled (burned, deleted, etc.) with common burn algorithms used for real flashes.
The simulated flash can be switched to RAM-Mode, which allows the user to use the flash as RAM, to have the possibility for fast copying (without the burn algorithms) of data to the flash memory area.

JTAG Interface Support for Intel® JTAG Cable

The Intel® XDB JTAG Debugger now supports all versions of the Intel® JTAG Cable (Rev. 1 and Rev. 2) additionally to the Macraigor OCDemon* Raven ARM* 20 JTAG interface.

The Intel® XDB JTAG Debugger startup dialog is extended with a new drop-down box labeled `JTAG Device`. This box contains 2 entries to switch between the Macraigor* OCDemon* Raven ARM* 20 JTAG interface and the Intel® JTAG Cable. These entries are:

```
Intel(R) JTAG Cable
OCDemon* Raven
```

The default selection is `OCDemon* Raven`.

JTAG Monitor Address Selection

Selecting the JTAG Monitor Address is now possible by entering the address into the `Monitor at:` input field of the Intel® XDB Debugger startup dialog / `CPU - JTAG tab`.

Simulator Instruction Count Based Profiling

The Intel® XDB Simulator Debugger provides the possibility to count machine instructions executed by the application. The functionality is available after enabling the Performance Monitoring Unit (PMU) of the simulated processor.

Peripheral Register Database

The Intel® XDB Debugger distributions include a tool set that allows customers to add new registers to the XDB framework. XML defined registers are now the preferred way to add register lists and windows to all XDB components.

Tools included are:

- Compliance checker for Intel® XDB specification of XML files.
- Converter for converting XML files to the Intel® XDB proprietary XRD format. This format allows fast loading of the different register descriptions.

Dialog History

The GUI of the Intel® XDB Debugger is extended with a history of string entries for dialog boxes. It enables dialog boxes to remember the last entered string in edit fields and will display them in drop-down list boxes in place of plain edit control in dialog boxes.

Syntax Coloring in Assembler and Execution Trace Window

This feature is a change in the appearance of the Intel® XDB Debugger GUI.

UART Simulation and Terminal Window

The UART simulation plug-ins provides the possibility to redirect the I/O stream to an internal debugger window. The Intel® XDB Debugger provides a terminal window that can be mapped to simulated UART devices. Characters sent by the UART will be displayed by this window, characters typed in while the window holds the focus will be sent to the UART device. The window additionally provides the possibility to emulate VT52 mode.

Loading Pre-booted Operating Systems in the Simulator

A new Plug-in for the Intel® XDB JTAG Debugger and the Intel® XDB Simulator Debugger provides the possibility to capture the complete state of a currently simulated system and store the information within a configuration file. This file then can be used in further sessions for fast reproduction of the loaded system environment. The main purpose of this feature is to save boot time in the debugging and simulation of operating systems.
The Simulator plug-in implements the following commands (that can also be executed through the GUI in interactive mode):

```
SIMS “SAVE <filename>
SIMS “SAVE”
```

```
    SIMS “LOAD <filename>
    SIMS “LOAD”
```

The two first commands shall save the full simulator state in the files `<filename>.xml\<filename>.bin` and `default.xml\default.bin`, respectively. The state of all simulator components (simulator core, memory and simulator plug-ins that support the feature) are saved in these files.

Module Window Improvements

The Intel® XDB Debugger Module Window is improved with columns and column headers as follows (including 3 samples modules):

Module	Symbol File	BD-ID	Status	Source Patch
Hello	`hello.bd`	1	loaded	`c:\xxx\hello.c`
Dyn	`hello.bd`	1	unavailable	`c:\xxx\dyn.asm`
Core	`os.bd`	2	loaded	`c:\os\core.s`

Note: The memory range information shown in the former versions of the Module Window has been removed. A click on the column header sorts the table by that column. The sort mode is not persistent. A newly opened window is always sorted by module name regardless which sort mode was previously active.

Usability of Evaluation Window improved

The usability of the Evaluation window in the Intel® XDB Debugger is improved by enabling inline editing of expressions.

Context Help for XDB GUI

The main Toolbar of the Intel® XDB Debugger GUI is extended by a Context Help button. The Context Help is displayed when pressing this button and clicking within a XDB window.

Shared Global Symbol Table

The symbol search in the Intel® XDB Debugger is extended to all global symbol tables of the loaded symbol files. This improves the Debugger usability when multiple symbol files are required. The global symbol table feature is controlled by the following new commands:

Option	Description
`SET OPTION/SYMBOL=GLOBAL`	Enables the shared global symbol table (default setting)
`SET OPTION/SYMBOL=LOCAL`	Disables the shared global symbol table
`SHOW OPTION/SYMBOL`	Shows the current state of the option

New Flash Plug-in Configuration Dialog

The `New Configuration` dialog of the Intel® XDB Flash Plug-in has a new drop-down field with a browse button for the selection of an XDB batch script. This allows to execute an initialization batch script before any flash operations that require an initialized target and makes the Flash-writer Plug-in independent from the startup batch file.

Shared Libraries Debugging

The Intel® XDB Debugger is now able to debug shared libraries which are already loaded on the target. This feature is supported by the following OS configurations:

- Linux* Kernel Mode (module debugging)
- Symbian OS*
- Palm OS*

RESTART Command Improved

The Intel® XDB Debugger command `RESTART ["file"]` has been expanded.
The steps performed with this command are as follows:

1. The Intel® XDB Debugger saves enabled and disabled code/data breakpoints temporarily and deletes them.
2. A restart of the target by asserting a CPU RESET signal is done.
3. The execution stops at the reset vector.
4. If a batchfile as parameter is specified it is executed now.
5. After execution of the batchfile (if any) all enabled/disabled code/data breakpoints are restored.

The purpose of a batch file is to execute any hardware configuration required directly after reset.
Typical usecase: After a reset of the target not all address lines are enabled for accessing the flash until a set of GPIO commands is executed. If breakpoints are set on flash addresses, the flash must be enabled with executing a batch after a restart prior to breakpoints restoration.

Line Table Support

The Intel® DWARF2BD Object Converter now produces a line table to support optimized code debugging. The line table is contained in a separate section and has no impact on the rest of the `*.BD` file contents.
Empty blocks are no longer written to the `*.BD` file. The filter analyzes the contents of a block and drops empty ones.

Location Lists for Global Variables

The Intel® DWARF2BD Object Converter now supports location lists for global variables.

General Linux* System and Application Debugging Support

Intel® XDB OS Awareness Plug-ins for the debugger are now available which allow Linux* system and application debugging.

The Plug-in for system debugging provides two windows: One window shows information about the Linux kernel threads, and the other window shows information about the loaded modules. The Plug-in for Linux system debugging connects to the target via JTAG interface.

The Plug-in for application debugging contains a Monitor running on the Linux target and a GPL kernel driver. The Plug-in for Linux application debugging connects to the target via serial interface.

Linux* 2.6 Kernel Support

The Linux* kernel 2.6 support is an add-on which permits to debug Linux* 2.6 kernel code and its modules.

Differences between Linux* 2.4 and Linux* 2.6 support

Process List information

The information displayed is the same as for the Kernel 2.4.

Relocation of the Debug information of the kernel modules

The Linux 2.4 support was consisting of reading the list of exported symbols to relocate the debug information of the kernel modules.
This operation was time consuming and is no more necessary with the kernel 2.6.

To generate the debug information of the kernel module a temporary library needs to be created from the kernel module (`.ko` file).
The procedure is the same as for the Linux 2.4:

```
arm-linux-ld –shared –g module.ko –o module.so
dwarf2bd –rel module.so –o module.bd
```

Note: Only the `.text` section can be relocated.
This has the following impact for the debugging capabilities:

- Accessing global variables defined in a kernel module is not possible

- Accessing code which uses a special link option can make the code inaccessible.
  Example:
  ```
  static void _init myinit()
  {
    ...
  }
  ```
  Puts the function `myinit` into another section than `.text`

  Workaround: The function `myinit` should call an auxiliary function which is in the `.text` section.
  Example:
  ```
  #ifdef DEBUG
  static void myinit_text();
  static void _init myinit() { myinit_text(); }
  static void myinit_text()
  #else
  static void _init myinit()
  #endif
  {
   ...
  }
  ```

If compiled with DEBUG flag set, the function `myinit` calls `myinit_text` which can be debugged in source code.

Accessing the Kernel modules code
The Linux* 2.6 protects the access of the kernel modules if these are not used. So the code and the data of a kernel module can not be viewed if these are not activated.
Impact, when the access to the modules is disallowed:

- The “Module List” window can not be updated.
- Setting software breakpoints in kernel modules is no more possible. In this case, it is necessary to use hardware breakpoints and modify it to a software breakpoint when a hit occurs.
  NOTE: The software breakpoints which were set into a kernel module remain valid.
- The debug information of kernel modules can not be relocated. One solution to get the debug information of the kernel modules relocated is to stop and continue the target just after the return of the function `sys_init_module` and `sys_delete_module`. This can be performed using a special watchpoint on the module list mutex:
  ```
  SET WATCHPOINT /ACCESS=WRITE /LENGTH=4 AT &module_c$$1\\module_mutex.count
  CONTINUE HARD 0
  ```

Starting debugging

The only changes between the Linux* 2.4 and 2.6 support for the usage is the additional watchpoint which needs to be setup in the startup script (use the configuration file `xdb\configurations\jtaglinux\jtaglinux26.xsf` which include this watchpoint)
The Linux plugin itself detects if the Linux 2.4 or 2.6 is debugged according the debug information file.

Symbian OS* v8.1b/v9 Build and Debug Support

A new command line option for the Intel® C++ Compiler and Intel® Linker, `-platform=symbian,` allows creation of Symbian OS* DLLs and executables.

The Intel® XDB JTAG Debugger allows debugging of Symbian OS* v8.1b and v9 code generated by GNU or ARM* RealView2.0/2.1 for Symbian OS* v8.1b/v9.

Access for Unloaded Modules in Palm OS*

Setting breakpoints in unloaded modules is now supported when debugging Palm OS*.

Debug Information Compatibility for GDB

A new compiler option `-Qvariable_locations-` can be used to improve compatibility of the debug information when interpreted by GDB. This option is recommended only when GDB is used for debugging instead of the Intel® XDB Debugger.

Example: `ccxsc -Od -Qvariable_locations- -Zi test.c`

FLEXlm* License Protection for Intel® XDB JTAG Debugger/Simulator

The installation process will check for the existence of a valid FLEXlm* license file, regardless of whether the complete Tool Suite is being installed, or any FLEXlm* protected part of it is being installed.

The following components of the tool suite require a FLEXlm* license file:

- Intel® C++ Compiler
- Intel® XDB Debugger
- OS Awareness Plug-ins for the Intel® XDB JTAG Debugger for:
  - Palm OS*
  - Symbian OS*
  - Linux*
  - Nucleus* OS

Defects Fixed in Version 2.0.1

This sections shows defects fixed with the update version 2.0.1

Tool	Description
Intel® Compiler	Missing "and" instruction for var++ in while loop
	Including <list> in a source file throws compilation errors.
	memset with unaligned pointer and length 0 cause endless loop
Intel® Assembler	Usage of options -j and -ccxsc-compatibility-model is not working
Intel® XDB Debugger	Wireless MMX instruction WUNPCKELB not working in Simulator.
	Disassembler does not show Wireless MMX 2 instructions
	Load instruction behaves incorrectly in Simulator when accessing uninitialized memory
	User created XDB buttons become inactive
Intel® DWARF2BD Object Converter	SDT1.2 outputs not converted correctly
	Sorting symbols takes a long time when debugging Linux
Intel® CodeWarrior* Plug-in	Changed paths in Stationary so path uses ‘intelxsc’ rather than ‘xscale’.
Intel® Integrated Development Environment	"Save as" message confusing if calling set options for debuggers in Workbench
	-I$(BASE)\path and -pd "<symbol> SETA <x>" not supported

Defects Fixed in Version 2.0

This sections shows defects which were fixed with version 2.0

Tool	Description
Intel® Compiler	Optimization issue with unsigned char types using the ++ operator
	Access violation using -Zi -GX -GR
	__global_reg(n) ignored
	The destructors of static members of template classes are not called at program termination.
Intel® Libraries	macro _HAS_NAMESPACE nowhere defined
	Namespace std has no member cos, cosh, log, log10, sin and sinh
Intel® Assembler	ASXSC-E-FATAL[0131]:Illegal storage access received
	Wrong address calculation on bl instruction
	Local Labels cannot be re-used in GNU Mode
	Invalid line numbers generated for sub macros
Intel® XDB Debugger	The SCAN command does not change the value of XDB variables.
	Save as message confusing if calling Set options for debuggers in Workbench
	Dis-assembler does not show Intel® Wireless MMX™ 2 instructions
	Intel® Wireless MMX™ 2 instruction WUNPCKELB not working
	Signed/unsigned type mismatch in Intel® Wireless MMX™ 2 instruction WADDBHUS
	Cannot evaluate static data members in classes
	nk.bin flashing fails
Intel® DWARF2BD Object Converter	Setting a breakpoint on a line returns wrong address (module offset)
	Breakpoint cannot be set if multiple source lines point to same address
	Incorrect handling of static variables
	Inlined symbol info not handled correctly
	Sorting symbols for minutes on end when debugging Linux*
	Multiple files with same name not handled correctly
Intel® CodeWarrior* Plug-in	Registry key for CodeWarrior* plug-in not created correctly

Known Limitations

Intel® C++ Compiler

Restrictions in C/C++ Language Support

- The ISO/IEC 9899:1999 (E) features are not fully supported
- Universal character set (Unicode) is not supported

GCC Features

The Intel® C++ Compiler doesn't support all features of GCC.

Deviation from ARM* EABI

Intel® SDT 2.0.1 does not support all ELF features. Linking objects from third-party tools might fail in rare cases.

Generation of Static Constructors

When using the options `-abi=rvct21` or `-abi=eabi1`, static constructors might not be executed by the generated program.

Workaround:
Within the linking stage, provide a linker build file (specified with the linker option `-buildfile`) that explicitly loads the `.init_array` sections to the generated image.

Example:
The build file should contain following lines:

```
LAYOUT
  SEGMENT CODE BASE 0x8000
  LOAD
    SECTION .text1
    SECTION _CODE_
    SECTION _RODATA_
    SECTION .init_array
```

Note:
Use option `-abi=rvct21|eabi1` only if you need to generate backwards-compatible code. For new projects don't specify an `-abi` option, since default option is `-abi=EABI2`.

Using `__declspec(dllimport)` and `__declspec(dllexport)` directives with optimization and C++ classes

There is a known issue when using the `__declspec(dllimport)` and `__declspec(dllexport)` directive in conjunction with optimization and C++ classes. The compiler may inline some of the constructors and incorrectly omit the instructions, even though the constructor has been declared as an exported function. The result is that the exported constructor is not present in the object file, which may cause problems in DLL or Shared Object linking at runtime or build-time. This problem only occurs with optimizations switched on in the compiler command line and does not occur with the `-Od` compile-time switch.

Internal error 0_0 when using `-Qipo_obj` with `-Qipo`

The Intel® C++ Compiler issues an internal error 0_0 when both `-Qipo_obj` and `-Qipo` compiler options are specified. The option `-Qipo_obj` implicitly uses option `-Qipo` so that there is no need to specify both.

Section attribute may be ignored for `comdat` functions

The Intel® C++ Compiler does not apply the section name attribute to class member functions.

C++ Exception Handling

Applications using C++ Exception Handling require full recompilation of the code using the Intel® SDT 2.0 release.

The Intel® C++ Compiler does not support goto outside a try/catch block with correct destruction of object created within the try catch block

Missing C++ Header Files

The following ANSI C++ header files as required by ANSI-C++ (ISO/IEC 14882 1st edition 1998-09-01) are not included in this release:

```
bitset
limits
locale
valarray
```

Makefile dependency file extensions

The output extension should be specified with `-o <output file>` and the `-QM` option should pick up this extension for the makefile dependencies. However, the output extension created for the makefile dependencies files is `*.obj`.

Workaround:
Manually change extensions in a dependency test file by replacing all `*.obj` occurrences with `*.o`.

Example:
After replacing `*.obj by *.o` extensions, run `ccxsc -QM test.c -o test.o`

Missing warning for explicitly aligned data structures on the stack

The Intel® C++ Compiler doesn't issue any warning or error message when the data alignment exceeds the guaranteed 8 byte alignment on the stack.
Example:

```
typedef struct __attribute__((aligned(16))) DMA {
int a,b,c,d;
} DMA;

extern void bar(DMA*);
int foo()
{
DMA x;    // should be 16 byte aligned, but can not be
bar(&x);  // bar requires to receive a 16 byte aligned data structure.
return x.a;
}
```

Option `-Qvec_report[n]` not supported

The Intel® C++ Compiler command line help describes a compiler option `-Qvec_report[n]` which however is not supported.

Miscellaneous

- When compiling anonymous unions, the compiler produces the warning:
  ```
  file.c(num): warning #40: expected an identifier
  ```
  You can safely ignore this warning.
- The structure elements of 64-bit data types (`double, long long`, and `_m64`) are not aligned to 8 bytes.
  For example, field d will be aligned on a 4-byte boundary:
  ```
  struct S1 {
  char c;
  double d;
  } gs1;
  ```
- Bitfields are currently not aligned. The size of the structure f1 should be 4, not 8:
  ```
  struct {
  char c1;
  int __attribute__((aligned(x))) b:1;
  } f1;
  ```
- The compiler currently does not perform according to the C++ standard 7.1.2, paragraph 4, sentence 6, stating that a string literal in an extern inline function should be the same object in different modules.
- The compiler does not emit any warning or error messages if a const object is passed as non-constant parameter to a function. For example:
  ```
  void f(char *c) { *c = 'a'; }
  int main() {
  const char *x = "b";
  f(x);
  }
  ```
  This does not happen in ANSI compatible mode (forced by the option –Za).

Intel® Assembler

PC relative addressing in PIC code issues with older Assembler versions

Older Assembler versions did incorrect calculations of some of PC relative addressing instructions and as a result workarounds may have been implemented in application code. Some of these workarounds are now unnecessary and should be removed; otherwise the workaround could themselves now cause the use of an incorrect pointer. If this is the case a data abort in run time could be the effect.

If you encounter a data abort in run time which is related to a PIC pointer please review the source to see if a workaround has been implemented. Typically such work around would with help of LDR/ADD instructions calculate the correct pointer from an earlier incorrect received result. An example of an instruction which now executes correct, but did produce incorrect results in earlier Assembler versions is: `add r10, pc, #LABEL -(.+8)`

Example:
In the following code sequence the pointer used in the `str` instruction will be different in the new Assembler compared with older versions. The first `add` instruction is now correct and `r10` will after this instruction point to `LABEL`. The next two instructions are 'workaround' needed earlier, but is now surplus and will produce a pointer `[r10]` which is different then expected.

```
    add r10, pc, #LABEL -(.+8)
    ldr r11, =LABEL
    add r10, r11, r10
    str r0, [r10]
```

Missing error message/warning on undefined symbols

Invalid/undefined constant in ASM instruction does not generate any error and results in incorrect code being generated.

Local Labels cannot be re-used in GNU Mode

The code example below produces an error message, which does not occur with the GNU ASM:

```
ASXSC-E-ERROR[0049]:ex1.s:15:Duplicated label ``, previously defined at line `8`
```

```
    @@@@ START OF CODE EXAMPLE
    .text
    mov r0, r0
    b 0f
    label:
    mov r0, r0
    mov r0, r0
    mov r0, r0
    b 0b
    label:
    b 0b
```

Miscellaneous

- Assembler cannot detect indirect recursion so it may go to infinite loop in macros, which uses this kind of recursion.
- Using listing file may not output the last line correctly if there is no new-line before the end of the file.
- Intel® Wireless MMX™ technology is enabled to work with ARM* style syntax only.
- The ARM* style `WHILE` directive may not work when there are label definitions inside the loop.
- Frame description directives are accepted but are ignored.
- The command line option `•D` to provide a define on command line is not supported anymore. However, there is another option which should be used for this purpose:
  `-Dmydefine=value` is equivalent to `-PD "mydefine SETA value"`
- The listing does not contain cross-references.
- The Intel® Assembler may generate an incorrect address for a conditional branch containing a variable.
- When macro1 is calling macro2 and macro2 is calling macro1 (endless loop), the Intel® Assembler exits with abnormal program termination and the operating system runs out of virtual memory.
- Exiting a macro with `.EXITM` before `.ENDM` may not work correctly.
- Defining a macro with wrong parameters does not always generate an error message; instead, the Intel® Assembler may go into an endless loop.
- The Intel® Assembler does no longer accept comments starting with a semicolon `;` in GNU mode. You have to replace the semicolons with `@` in the source code.

Intel® Linker

Position-independent Code and Data (PIC/PID)

If the application contains initialized data in PIC/PID-code, the compiler has to be called with the command line option `-init DATA` by using the function `__inixsc_init` in the code.

The options for PIC/PID are `•ropi` and `•rwpi`, as documented in the Intel® Linker User's Manual.
Note: The synonymous options `•pic` and `•pid` are no longer supported by the Intel® Linker.

Please refer to Initialization for ROMable and PIC/PID Code in the Documentation Addendum for a detailed description.

Miscellaneous

- The command line options `-robase` or `-rwbase` cannot be used together with `-buildfile`.
- If a customer uses a build file without explicitly loading all the BSS sections in the layout block (i.e., without the command `load section _bss_`) then it is possible that common symbols of the application cannot be relocated accordingly.
  Note: The compiler emits common symbols for all non-initialized global variables.
- If an alias for a global label is defined in the build file, the symbol table of the output file shows this alias as "undefined". However, all references to the symbol are resolved correctly, and the resulting code is correct. Please ignore the misleading information in the symbol table.

Intel® Libraries

`<cwchar>` is not compatible with `-D_HAS_NAMESPACE` (C++ STL Library)

When the `_HAS_NAMESPACE` macro is defined, some use cases may fail, reporting numerous undefined symbols.

The following code fails:

```
#include <stdio.h>
#define _HAS_NAMESPACE 1
#include <cwchar>

int main() { return printf(""pass\n""); }"
```

`errno` not set in Math Library

The following math library functions do not set `errno` as mandated by the ISO/IEC 9899:1999 (E) international standard:

```
lrint
llrint
lround
llround
lroundf
llroundf
ldexp
scalbn
scalbln
```

For the following math library functions `errno` may not be set correctly as mandated by the ISO/IEC 9899:1999 (E) standard; additionally there may be round-off issues:

```
    fdim
    fdimf
```

Intel® XDB Debugger

Incorrect use of terms `Word`, `Long`, `Long Long` in Debugger GUI

The selections for `size` in the context menu of the memory window of the Intel® XDB Debugger do not match the conventions for the ARM* Architecture.

Refer to this cross-reference to match the right ARM* conventions:

Byte Size	XDB Naming	ARM* Convention
2	Word	Half Word
4	Long	Word
8	Long Long	Double Word

Example:
Open a memory window. It displays `size` default as Byte (2-hex digits). Open the context menu and switch `size` to `word` (2-Bytes) which is a Half-word in ARM* convention.

Directory name in dialog box too long

If the debugger cannot find a source file, it opens a dialog box to allow the user to search for the specific file. However, if the relative path shown in the dialog box title is too long it is difficult to find the correct file.

Possible workarounds:
- Relevant source directories can be specified using a XDB batch file at the beginning of the session whereas the dialog box will never appear.
- The debug info filter can be used to substitute the relative path with a full path to file.

Debug line information missing on closing brace `}` following a `return` statement

Example:

```
     test(){
       if (x)
       return 0;
     } // debug information will not be generated here
```

Debug line information missing at `break` statements

Debug line information is missing at `break` statements. This happens in various looping structures and `switch` statements.

Inline Assembler support

- Branches exceeding their respective limits might not raise an error and the instruction might be assembled wrongly.
- Pseudo-instructions like `ADRL` or `LDR` are not allowed and if used might yield wrong assembly.
- Only instructions are expected. Use of other directives may result in unexpected errors.

Register Descriptions in Bitfield Editor

The Bitfield Editor of the Intel® XDB Debugger shows descriptions of the various registers. Although these descriptions are being updated regularly, it is possible that some of them are not up-to-date. Please refer to the latest version of the Developer's Manual for the target processor.

Miscellaneous

- The default format of double numbers used in the Intel® XDB Debugger does not match the Intel® C++ Compiler format. To switch to the compiler double number format, use the following debugger command:
  XSCPU "SWAP DOUBLE"
  We recommend adding this command to your initial batch file so that the correct format is automatically displayed.
- If the user changes the mode of the last open MDI window to Docking Window and then back to MDI Window, then the Window menu does not work anymore until the XDB is restarted.
  Workaround: Always have one window in MDI mode, for example, the Command window or any other window that you usually do not close.

Intel® XDB JTAG Debugger

Hardware Breakpoints

The Intel® XDB JTAG Debugger for Intel® JTAG Cable debugger supports hardware breakpoints that are stored in the internal MINI-I-cache. This extends the two processor hardware breakpoints up to a higher amount of hardware breakpoints. This is a very useful feature if you work in a flash area, where software breakpoints cannot be used. However, these breakpoints have certain functional restrictions. These special breakpoints should not be used while paging is changing. Therefore this feature is disabled by default. It can be enabled by using the argument `-target MIMIIBP` in the `Additional Arguments:` field of the Intel® XDB JTAG Debugger startup dialog (`Intel(R) XDB Debugger - Startup / Settings / CPU - JTAG`). If enabled the debugger will output a warning when started.

The usage of these extended hardware breakpoints in a real paged environment (non 1:1 paging) is more complex. The breakpoints can only be used as long as the paging and code stays static. This is because the overlaying cache lines contain 32 bytes that are not affected by page changes and therefore would still show the old assembly instructions for these addresses. A typical problem in a paged environment is that you want to restart the application and keep your breakpoints. A restart will remove the paging and thus the extended hardware breakpoints will not work correctly. To restart your paged application and keep the breakpoints you can use the following sequence:

> *disable breakpoint /all*
> *restart*
> *run until <paging active>*
> *enable breakpoint /all*
> where *<paging active>* is a code location where the paging is completely set up.

Connection Problems with the Intel® JTAG Cable

The Intel® JTAG Cable does not support the speed parameter `LPT1:1` that follows the port name in the debugger startup dialog and ignores it.

The speed parameter can be modified in the `JTAG.ini` configuration file. The parameters are:

```
    highspeed=yes|no
    slowdown=<microseconds>
    hsslowdown=<microseconds>
```

The default settings are:

```
    highspeed=yes
    slowdown=0
    hsslowdown=0
```

It may happen that the driver for the Intel® JTAG Cable is not installed correctly.
If the Intel® JTAG Cable is not working, try to start the driver parbinst.exe manually as follows:
Please check if the driver `parbstone.sys` in the Windows driver directory (default: `C:\Winnt\system32\drivers`) is up-to-date, that is if it is the same as in `C:\Program Files\Intel\SDT2.0\xdb\tci\intelxsc\`. If not, copy the `parbinst.sys` from the `\xdb\tci\intelxsc\` to `C:\Winnt\system32\drivers` and start the `parbinst.exe` from `\xdb\tci\intelxsc\`.

Communication Speed Settings with the Macraigor OCDemon* Raven ARM* 20 JTAG interface

If a connection to the target board with the Macraigor OCDemon* Raven ARM* 20 JTAG interface is not possible, connect with connection speed of LPT1:4. Observations showed that after one successful connect with LPT1:4 it is possible to connect with higher speed LPT1:2 without any problems. The number after the colon is a divisor, i.e., a bigger number causes lower speed.

Reset exception

The Reset exception cannot be switched off, but disabling the SoftInt will print a warning that this disables the SWI handling. If the SWI plug-in is not loaded the SoftInt will not be captured by default.

Debugger crashes if started without .xsf file or targettype specified

The Intel® XDB JTAG Debugger crashes if option `-B` is passed without parameter.

If the Intel® XDB JTAG Debugger is started from the command line with options but without specified `*.xsf` file or target type, the debugger crashes.

Workaround:
Specify either an `*.xsf` file with the option `•xsf filename` or a target type with the option `•tgttype <targettype>`. The target type must be one of the entries that can be seen in the start-up dialog, e.g., Intel(R) JTAG Cable.

Breakpoints are not re-enabled after flash programming

When burning flash, active breakpoints are disabled to avoid them to interfere with the flash burning algorithm. After burning, breakpoints most likely are at wrong places and need to be evaluated. Especially breakpoints that were in the range of the burning algorithm are useless afterwards. Therefore after flash burning breakpoints are not re-enabled.

A workaround might be:
Use the debugger command `save /breakpoint` to save breakpoints before burning flash and `restore / breakpoint` to restore them afterwards. But please note that breakpoints may be on wrong places after flash burning!

Registers not writable by JTAG using the Intel® JTAG Cable

The following two XDB commands lead to target error in the Intel® XDB Debugger with the Intel® JTAG Cable:

```
xdb> set sysreg cache_inv_i_btb=0xffffffff
E-L-C-SSYR : set system register 'cache_inv_i_btb:' target error : Register: not
writable by JTAG

xdb> set sysreg cache_inv_i_d_btb=0xffffffff
E-L-C-SSYR : set system register 'cache_inv_i_d_btb:' target error : Register:
not writable by JTAG
```

The following commands do NOT lead to errors:

```
xdb> set sysreg cache_inv_btb=0xFFFFFFFF
xdb> set sysreg cache_inv_d=0xFFFFFFFF
```

This is a security feature and can not be changed.
The Intel® XDB JTAG Debugger using the Intel® JTAG Cable supports breakpoints in MINI-I-cache. Therefore the debug handler runs in I-cache while the target is stopped. Invalidating the entire I-cache by the commands mentioned above would erase the currently running debug handler and the connection would be lost.

Note: With the Macraigor OCDemon* Raven ARM* 20 JTAG interface this error does not occur, that is the registers are writable by JTAG since Raven doesn't use breakpoint in the MINI-I-Cache of the processor.

Intel® XDB OS Awareness Plug-in for Palm OS*

Exception Vector setup

After starting the debugger, Palm OS* can be booted normally. Please note that the exception vectors need to be set up in the Mini-I-Cache.
After Palm OS* has booted, the Palm OS* plug-in needs to search for the kernel object information. To trigger the search process, type the following command into the Command window:

```
PALMOS "RESCAN"
```

After a short while, the Palm OS* windows in XDB can be opened, displaying the according kernel object information.

For support of the position-independent code in Palm OS*, the debug loader hooks need to be installed manually. Typically they are at address 0x00001000 or at address 0x510FF000, depending on how the BigDAL is generated. Please refer to your BigDAL build time options for selecting the appropriate address.

You can use the following commands to install the hooks:

```
   set val *(unsigned long*)0x1008=0xE1200F7E
   set val *(unsigned long*)0x1018=0xE1200F7F
```
or
```
   set val *(unsigned long*)0x510FF008=0xE1200F7E
   set val *(unsigned long*)0x510FF018=0xE1200F7F
```
These commands should be executed before the BigDAL is booted, but after the MMU has been activated. You can stop execution at the start of the BigDAL by setting a breakpoint at address 0x20040000 using the following command:
```
   set breakpoint at 0x20040000 hard
```
This command should be used immediately after the debugger has started, to avoid missing the entry into the BigDAL.

ROM protection in DBPXA25x BSPs

All current ports of Palm OS* to the DBPXA25x have the ROM protection in the page tables disabled. This prevents the debugger from selecting the appropriate type of breakpoint for code in flash memory. The current recommendation is to use only hardware breakpoints. For instance where it is certain that code is in RAM, the option to use software breakpoints can be enabled. See also SET OPTION /HARD in the Intel® XDB Debugger Reference Manual.

Intel® XDB OS Awareness Plug-in for Symbian OS*

Incorrect values displayed

The values displayed for the Run Address, Home Address, Code Size, Const Data Size and BSS Size of a Process as well as the values displayed for the Code Address, Code Size, Constant Data Address, Data Run Address, Data Home Address, Data Size, Bss Size and Constant Data Size of a Library may be incorrect, depending of the OS version currently being debugged.

Debug Information and Files with the `.cia` Extension

Using the ARM* RVCT 2.1 compiler for files with `*.cia` extension invokes a special build process. The original file is compiled using a three step process. First, it is preprocessed by the compiler. Then the GNU style syntax of the inline assembler statements is converted to ARM* style syntax. Finally, this temporary file is passed back into the compiler.

During this process, the line number information gets lost. The compiler creates the debug information for the temporary file, which is different from the original `*.cia` file. Thus, the debug information is incorrect.

If such debug information is loaded, the Intel® XDB Debugger produces a `line-not-block` error message.

Intel® XDB OS Awareness Plug-in for Linux*

System Mode/Device Registers not accessible

System mode registers such as the coprocessors CP14, CP15 and the Translation Table register are locked by the system.

Device registers are controlled by the Linux* OS and therefore direct access to these devices is prohibited. You have to use I/O commands to access device registers.

Nevertheless, all these registers are visible by the Intel® XDB Debugger windows with the Intel® XDB OS Awareness Plug-in for Linux* Application Debugging loaded, but they show dummy values and all contents of these windows is meaningless.

Debugger does not start `dwarf2bd` when blanks in the path to `vmlinux`

If the path where `vmlinux` is located contains blanks, the Debugger cannot start the object converter DWARF2BD.

Kernel modules can only be debugged if `Kernel Modules` window is opened

It is only possible to debug kernel modules (Kernel 2.6) if the `Kernel Modules` window has been opened before loading the modules.

Linux* fails to boot if Debugger started with MMU setup script

Linux* doesn't boot on target while the Intel® XDB JTAG Debugger is running if debugger was started with MMU setup scripts.

Workaround:
Don't use any debugger script except of (default) `C:\Program Files\Intel\SDT2.0\Example\ocdxs\batch\exoff_linux.xdb`.

Intel® XDB Simulator Debugger

User-defined keys

User-defined keys do not work properly in some windows.

Issues on extensive UART usage

The Intel® XDB Simulator Debugger might hang when simulating codes that make extensive use of UART (as operating systems do, for instance). The issue might emerge when the code (e.g., a serial driver) makes an attempt to restart or reconfigure the communication settings, just after a character has been written or received in the FIFO. It is not possible to recover from this issue yet, and the simulation session has to be closed.

No calibration of touchscreen

The touchscreen simulator plug-in does not support all working modes present on DBPXA250 and DBPXA27X boards. In few configurations, values reported by analog-to-digital converter (ADC) are out of range. It is in particular the case with Palm OS* and Symbian OS*. It is therefore not possible to pass calibration steps for these operating systems.

Load instruction behaves incorrectly when accessing uninitialized memory

A load instruction with uninitialized memory behaves incorrectly since uninitialized memory is in an undefined state.

Therefore it is important to initialize memory prior to any load instruction in the Simulator.

Intel® DWARF2BD Object Converter

Artificial name `__unnamed_block__address` used for labels in memory disassembly

In memory disassembly the artificial name `__unnamed_block__`*`address`* may appear created by the Intel® DWARF2BD object converter. The label name doesn't appear in the debug info or the ELF symbol table. To have the correct name, an `EXPORT` *`label`* statement has to be inserted in the `*.s` file to have the correct name in the debug info. However, this doesn't affect the functionality of the debug info.
Example:

```
SECTION 0: 0x0001 0x000080ac __unnamed_block__80ac TYP F,V 0x00
SECTION 0: 0x0001 0x000080ac __unnamed_block__80ac PRC <28,0>
```

The address `0x000080ac` points to a label `_start:` in the source file. Instead of label name the name `__unnamed_block__80ac` is shown.

Workaround:
Insert `EXPORT _start` into the `.s` source file and you will have the correct label name in the disassembly:

```
SECTION 0: 0x0001 0x000080ac _start TYP F,V 0x00
SECTION 0: 0x0001 0x000080ac _start PRC <28,0>
```

Intel® XDB Flash Memory Plug-in

Update of old Configuration Files

The current version of the Intel® XDB Flash Memory Plug-in cannot load flash configuration files (*.fcf) created by versions before 2.01.
Note: To see the version of the plug-in, start the Intel® XDB Debugger and select File / Show Plug-ins.

If you have configuration files created by an older version of the plug-in, you have to renew the configuration files. Perform the following steps:

1. Open one of the sample configurations `*.fcf` that fit to your board by using the menu item `Flash / Open configuration`.
2. Select the menu item `Flash / New configuration`.
   You will see the preset configuration copied from the sample configuration; therefore most of the displayed data is already correct.
3. Browse for the data file you intend to burn. If your image used a special offset, type it into the `Data File Offset` field. The new burning algorithm can be moved to any RAM area by using the Load Burn Algorithm to field. It is usually not necessary to change this value.
4. Save your configuration using the name of your old configuration.

Intel® Flash of B-Type does not support CFI

The Intel® Flash of B-Type do not fully support the Common Flash Interface (CFI). They also do not support locking and unlocking.
These flashes can be burned and erased with the Intel® XDB Flashwriter Plugin, but the Overwrite button must be set for burning and the lock and unlock functions can't be used. The flash is always unlocked.
If the JEDEC-ID check returns 34966 (0x8896), you have to enter the flash type manually into the database.

Intel® XFlash

The Intel® XFlash cannot burn an image file with an offset of 1, 2, or 3.

Intel® IDE

Renaming workbench files `*.nwp`

A workspace file `*.nww` can contain multiple workbench projects `*.nwp`. If you rename a project `*.nwp` you must take care to rename also the corresponding project name in the workspace `*.nww` file; otherwise when closing the workspace from the Intel® IDE, the original `*.nwp file gets` overwritten with contents of the new `*.nwp` project file.

Missing project templates for Intel® PXA2xx/PXA9xx processors

The New Project Template dialog from the Intel® IDE contains a template for the Intel® PXA800F communication processor only, but not for the Intel® PXA25x, Intel® PXA26x, Intel® PXA27x and Intel® PXA9xx processor families.

Workaround:
Use any other Intel® IDE Project Template and modify it accordingly by replacing debugger startup and configuration scripts in the project window.

Workbench linker options dialog produces error on default library listing

Linker settings in workbench Build Manager:
```
-listing=$(PROJECT).map -output $(PROJECT).elf ""$(SDT)\intelxsc\lib\angelswi
\x0__a000.o"" $(INPUT) ""$(SDT)\intelxsc\lib\x0__ac00.a"" ""$(SDT)\intelxsc\lib
\angelswi\x0__as00.a
```
Click on **`Set Options`** produces warning:
```
Unknown Option: ""$(SDT)\intelxsc\lib\angelswi\x0__a000.o"" $(INPUT) ""$(SDT)
\intelxsc\lib\x0__ac00.a"" ""$(SDT)\intelxsc\lib\angelswi\x0__as00.a
Do you want to keep these options Yes/No"
```

Old Projects using different macro name

The Intel® Integrated Development Environment macro `$(SDT)` contains the installation path to the tools of the Tool Suite. If you have old projects using a different macro name, the following error message will appear whenever a tool of the Tool Suite is to be activated:
```
  Unable to replace Macro in Tool toolpath
```
where `toolpath` is the installation path of the tool.

Perform the following steps to update the installation path macro:

1. Open the Build Manager dialog.
2. For every entry in the toollist describing a tool of the Intel® C++ Software Development Tool Suite Professional 2.0, correct the value in the Path field and in the Options field by replacing the incorrect macro name with $(SDT).
3. Close the Build Manager dialog.

Drag&Drop not working between active and non-active projects

Drag&Drop is only available within the active project in the Workspace. If you try to drag and drop between active and non-active projects, strange GUI behavior may result.

Cannot open files on non-active projects

You cannot open files of non-active projects until you activate the project.

CodeWarrior* Plug-in

Include Paths can be lost

When switching between projects in the Metrowerks* CodeWarrior* IDE, the include paths of projects can be lost. An error `Could not find source file` <*name*> is issued.
Workaround: After having switched to a project you want to build, open it's project settings and close the opened dialog with OK. Then you can build the project.

Wrong options in `Code Generation` panel

There are different errors in the `Intel Compiler Settings > Code Generation` panel,
for example:

- Option `-Qlfist` is undocumented and not supported by the compiler
- Option `-Qfnsplit-` is a undocumented option, but the compiler doesn't complain about it
- Options `-QA-` and `-Qfnsplit-` are passed without the trailing "-" to the compiler and thus they don't have any effect

Error messages from code generating tools not issued in Diagnostics Window

The Plug-in does not search for error messages issued to `sdtout` and `stderr` by the code generating tools. It just displays the message `Error: The object file was not produced` in the diagnostics window.

Wrong Path for Mapfile

The file `start.xdb` contains a wrong path to dgbmapfile. You have to change this path manually. The file `start.xdb` is located in the folder debug of the `IntelXSC_PNO\debug` sample project.

Replace the line
```
   palmos "set dbgmapfile DebugAlias.txt"
```
by the following line
```
   palmos "set dbgmapfile .\debug\DebugAlias.txt"
```

Limits in Dialog Fields

The following size limits (in bytes) apply to the entries of dialog fields:

Tool	Dialog	Size (Bytes)

Intel® C++ Compiler (general)	version text	500
	additional options	500
Intel® C++ Compiler (Preprocessor)	include file	4000
	include directories	4000
	undefine	4000
	define	4000
Intel® Linker	ctortab	50
	dtortab	50
	ctorpattern	50
	dtorpattern	50
	cdtorseg	50
Intel® Linker Libraries	libraries	500
	addopts	1024
Intel® Linker Output	output name	1024
	entry	1024
	base	1024
	exportOpt	1024
	buildfile	1024
	buildmacro	1024
	verbose	1024
	listing	1024
Intel® Library Manager	output name	1024
Intel® Object Converters	boundversion	50
	substitutedir	500
	verbose	50
	additional options	1024
Intel® Assembler (general)	output file	1024
	errorfile listing	1024
	additional options	1024
	additional options	1024
Intel® Assembler (listing)	listing file	1024
	via	1024
Intel® Assembler (preprocessor)	include	4000
	predefine	500

Warning: If an entry exceeds its size limit, no warning or error message will be issued. The entry is truncated silently.

No rebuild after changing compiler options

Changing compiler options is not noticed by the CodeWarrior* Plugin. You'll have to delete manually the previous generated output files in order to get the plugin to recompile the project (this works for changing linker options).

Miscellaneous

- Help messages for panels are not implemented.
- CodeWarrior* environment variables (Target / Runtime Settings in the preferences) cannot be used by plug-ins. The same is true for other settings of this panel.
- The provided file config_sim.xsf must not be renamed, because this file is necessary for some internal checks and routines.
- Do not use the Intel® C++ Compiler options -QCVasm_opt-mcpu or -QCVasm_opt3 (or similar options) to set the assembler CPU anymore. These options are obsolete for the Intel® C++ Software Development Tool Suite Professional 2.0! The compiler now sets the assembler CPU automatically according to the option •Qxscn. If you use the options •QCVasm_opt-mcpu or -QCVasm_opt3, the object cannot be generated with CodeWarrior* and the following error message is produced: “No object file is generated".
- One new plug-in was added. The Intel® Batch File Runner is able to execute batch files that were added to a project. The tool was tested with the Palm* Project (WholeROM.mcp) and is able to execute the batch files that are part of this project. However, there is no guarantee that all already existing batch files are executed successfully.
- Intel batch file runner doesn't support '(' and ')' in path and file names.
- The name of the Intel® C++ Compiler in the Target/File Mappings has changed to Intel(R) C++ Compiler. In projects created with previous versions of the Intel® CodeWarrior* Plug-in, the File Mappings should be hand-modified via the Target Setting Panel.
- Diagnostics on STDERR are ignored by CodeWarrior* Plugin.

Documentation

You can view the HTML-based product documentation with your Web browser, which provides full navigation, index look-up, and hyperlink capabilities. The getting-started documents and other documents also have PDF versions for easier printing.

A product documentation index is provided for easy access of all the documents. By default its location is`: C:\Program Files\Intel\SDT2.0.1\Doc\doc_index.htm,` where the name of the path, `C:\Program Files`, may be different depending on your Windows* operating system.

Documentation Errata

Global Register Variable Feature / Keyword not documented

The supported ARM* standard keyword `int __global_reg(n)` is not documented in the Intel® C++ Compiler Manual.

Syntax: `int __global_reg(n)`

where `n` is an integer with values:

# for n	Register
1	r4

2	r5
3	r6
4	r7
5	r8
6	r9

This is an important feature when combining PIC/PID and non-PIC/PID builds, since this Global Register Variable can be used to prevent the compiler from using r9 as a standard register.

`#pragma align` not Supported

This pragma is documented in the Intel® C++ Compiler Manual, but it is not supported by the Intel® C++ Compiler.

Wrong Syntax for Intrinsics `_mm_align_si64(), _mm_getwcx(), _mm_setwcx()`

The Intel® C++ Compiler User's Manual describes the following Intel® Wireless MMX™ Compiler Intrinsics incorrectly:

`__m64 __mm_align_si64(__m64 m1, __m64 m2, int count)`
should read
`__m64 _mm_align_si64(__m64 m1, __m64 m2, int count)`

`int __mm_getwcx(int number)`
should read
`int _mm_getwcx(int number)`

`__mm_setwcx(int number)`
should read
`void _mm_setwcx(int number)`

As for all intrinsics there is only one underscore in front of the function name.

Compiler Intrinsic `_mm_align_si64()` doesn't support `WALIGNR` instruction

Table 19-7 of the Intel® C++ Compiler User's Manual lists the Intel® Wireless MMX™ instructions `WALIGNI/ WALIGNR` as being used for the Compiler Intrinsic function `_mm_align_se64`. However the intrinsic uses `WALIGNI` only and allows passing the offset parameter as numeric value only, not as variable.

The syntax description

```
__m64 _mm_align_si64(__m64 m1, __m64 m2, int count)
```

in chapter 19 however is correct. It states that parameter `count` has to be a numeric value or expression that can be evaluated at compile-time and that it cannot be a variable.

Addendum - Initialization for ROMable and PIC/PID Code

The Intel® C++ Compiler User’s Manual and the Intel® Linker User’s Manual do not document the initialization mechanism for ROMable and PIC/PID code. The following sections describe this feature.

A general problem within the embedded world is that there is no loader to initialize the data section. For the position independent code and data (also known as PIC and PID) there is an additional problem for C++ with the pre-initialized C++ data structures (e.g. virtual function table and exception handling information). The following sections describe a solution for this problem. The information is generated by the final link step.

How to use the initialization mechanism with Intel® SDT

The linker is capable to generate the init data section if the option `-INIT <segment-name>[,-]` is used. This will translate these segments into one init data segment.
The function `__inixsc_init` should be called before any pre-initialized data is used.
The `__inixsc_init` function requires three parameters.

- The first parameter is the pointer to the init data section (e.g. `_seg_init_data_beg_`)

- The second parameter is the base address of the default data pointer (0 in an absolute linked example, otherwise the base address of the data section, e.g., R9 for position-independent data).

- The third parameter is an array of the different base addresses for the different sections. This is not required for absolute linked application, but for position independent applications to determine the destination base address and for relocations information the base address of the destination section.

How to invoke the initialization mechanism in the Linker

The option "`_INIT DATA,BSS`" is the option which should be used for default absolute applications. If the AngelSWI system dependent library is used, the call to `__inixsc_init` is already there. If an own startup is used, the following code should be added before any initialized data are uses:

```
LDR  R0, =_seg_init_data_beg
MOV  R1, #0
LDR  R2, =NULL_TABLE
BL   __inixsc_init
…
NULL_TABLE: .word 0,0,0,0,0
```

If the stack is with the DATA or BSS segment, the function `__inixsc_init_destroy_register_r4_r5_r6` should be used, since R4 to R6 are destroyed anyway.

Description of the `init_data` section contents

The linker has to generate one `init_data` section which has to be combined to the current code section.
In this case, the start address can be calculated relative to the current code address. The "`init_data`" section may contain the DATA and BSS initialization as well as the initialized data for the data segment. An optional relocation table should follow these records. The end record is mandatory for the "`init_data`" section. This mechanism also can be used to copy relocatable code into RAM. All self–relative relocation has to be resolved while section-relative relocation has to be added to the relocation table at the end of the "`init_data`" section. The section index is an 8-bit value. The value 0xff (==255) marks that the section base address is 0.

The linker should generate the following data structure to support ROM-able code and PID data. Because of the Intel XScale® technology structure, all records have to be aligned to the next 4-byte boundary:

The first long value is split into two fields:

Bit 31 – Bit 24	Bit 23 – Bit 0
`kind`	Additional information

Value of `kind`	Description
`0x00`	Plain Data Definition (8 bit)
`0x01`	Plain Data Definition (32 bit)
`0x02`	Fill an 8 bit area
`0x03`	Fill an 32 bit area
`0x04`	Set base address
`0x05`	Add relocation information (section relative)
`0x06`	Define Public Symbol *)
`0x07`	Define External Symbol *)
`0x08`	32-bit Section Relative Relocation *)
`0x09`	Self relative fixup *)
`0x0A`	Specify de-compress function for plain byte data
`0x0B - 0x0F`	Free for extension
`0xFF`	End of DATA section

*) only required if some dynamic linked library (DLL) or shared object (SO) functionality has to be supported

Description of the different Record Types

Kind 0xFF: End of Init Data

The remaining bits are free and unused (should be zero). If the initialization algorithm reaches this value, it stops and returns.

Kind 0x00 (0x01): plain 8 (32) data record

The remaining bits contain the number of data values that have to be copied (8 or 32 bit). If 32-bit data has to be copied, the address has to be 4-byte aligned.

Record Description:

1st Word [31:24]	1st Word [23:0]	2nd Word	N Words
0x00 / 0x01	Number of elements	Section offset	Data

Kind 0x02 (0x03): iteration record for 8 (32) bit data

The remaining bits contain the number of iterations for the data values that has to be copied (8 or 32 bit). If 32-bit data has to be copied, the address has to be 4-bytes aligned.

Record Description:

1st Word [31:24]	1st Word [23:0]	2nd Word	N Words
0x02 / 0x03	Repeat count	Section offset	Data

Kind 0x04: set new destination section

The remaining bits contain the new section index. This is an index into a section base address table that has to be provided to the algorithm. The pre-initialized value is the section index for the data section. With this, a new base value is loaded into the data base address register.

Record Description:

Bit 31 – Bit 24	Bit 23 – Bit 0
0x04	Section index

Kind 0x05: section relative relocation

The remaining field is split into two sub fields. The upper 8 bits specify the source section index for the fixup relocation. The lower 16 bits specify the number of relocations. The destination section has to be specified with the 0x04 record or the default is used. The linker has to generate an entry for each section relative fixup to a relocatable section. This fixup does not support relocations to unresolved externals. Therefore, the record 0x08 and 0x09 has to be used.

Record Description:

1st Word [31:24]	1st Word [23:16]	1st Word [15:0]	N Words
0x05	Source section index	Number of relocations	Offset within the data section

Kind 0x06: Define Public Symbol

The remaining field is split into two subsections. The first one is the public symbol section; the second one contains the number of public defined within this record.

Record Description:

1st Word [31:24]	1st Word [23:16]	1st Word [15:0]	N Words
0x06	Public section index	Number of publics	Offset within the section and the public symbol name

Kind 0x07: Define External Symbol

The remaining field is split into two subsections. The first is not used; the second contains the number of external symbols.

Record Description:

1st Word [31:24]	1st Word [23:16]	1st Word [15:0]	N Words
0x07	Free	Number of externals	External symbol name

Kind 0x08: External section relative relocation

The remaining field is split into two subsections. The first is not used; the second contains the number of relocations.

Record Description:

1st Word [31:24]	1st Word [23:16]	1st Word [15:0]	N Words
0x08	Free	Number of relocation	Offset within the section and external index

Kind 0x09: External self-relative relocation

The remaining field is split into two subsections. The first is not used; the second contains the number of relocations.

Record Description:

1st Word [31:24]	1st Word [23:16]	1st Word [15:0]	N Words
0x09	Free	Number of relocation	Offset within the section and external index

Kind 0x0a: de-compress function for plain byte data

The remaining bits contain the new section index. This is an index into a section base address table that has to be provided to the algorithm. The pre-initialized value is the section index for the data section. With this, a new base value is loaded into the data base address register.

Record Description:

Bit 31 – Bit 24	Bit 23 – Bit 0	4 Bytes
0x04	Section index	Function address

Function Prototype of the Initialization Function

int __inixs_init(unsigned long *InitDataBlock, unsigned long **SectionOffset, unsigned long *DefaultDataBasePointer);

int __inixs_init_destroy_register_r4_r5_r6(unsigned long *InitDataBlock, unsigned long **SectionOffset, unsigned long *DefaultDataBasePointer);

Return Value	
0	Reach the end
!0	An error occurs

The first parameter specifies the start address of the initialization block start address. The second parameter is either NULL or a pointer to the section base address table. For PIC/PID code, this may not be NULL and the table should contain at least two entries. The first entry is the CODE base address while the second one is the data base address. If the code is no PIC/PID code, the corresponding entries should be 0. The default data base pointer is the same like the PID data base address.

Example for ABSOLUTE or ROMable Code

SectionOffset = NULL
DefaultDataBasePointer = NULL

Data Stream	**Description**
0x03000450 0xa0100000 0x00000000	Fill 0x450 longs with the value 0x00000000 started at 0xa0100000
0x03002000 0xa0101800 0x00000000	Fill 0x2000 longs with the value 0x00000000 started at 0xa0101800
0x01000030 0xa0100034 <30 long>	Copy the next 0x30 longs from the current data stream address to 0xa0100030
0xff000000	End of the init data block

Example for PID init Section

SectionOffset[0] = CodeBaseAddress
SectionOffset[1] = DataBaseAddress
DefaultDataBasePointer = DataBaseAddress

Data Stream	**Description**
0x03000450 0x00000000 0x00000000	Fill 0x450 longs with the value 0x00000000 started at 0x00000000 + DataBase (default)
0x03002000 0x0001800 0x00000000	Fill 0x2000 longs with the value 0x00000000 started at 0x00001800 + DataBase (default)
0x01000030 0x00000034 <30 long>	Copy the next 0x30 longs from the current data stream address to 0x00000034
0x05000001 0x00000040	Relocation at DataBase + 0x40: read long and add the CodeBase and store back the relocated address value
0x05010002 0x00000048 0x0000005c	2 Relocation at DataBase + 0x48 (and + 0x5c): read long and add the DataBase and store back the relocated address value
0xff000000	End of the init data block

Now, the linker generates tables for the CODE and DATA sections at the end of the init data block. This also enables C++ and virtual function tables and exception handling. The linker generates the relocation table at the end of the init data block. This prevents the init data blocks from overwriting the pre-initialized data. The algorithm itself defines the relocation position. If the address is not aligned, the 32-bit base value has to be read from the byte stream, the section base address has to be added and stored back within a byte stream.

The following assembler source is the corresponding algorithm to unpack and relocate the data portion:

```
   @@
      @@ $Source:$
      @@ $Revision:$
      @@
      @@ This function is analyzing the ini section area and
      @@ expanding it to
      @@ initialized data regions
      @@
      @@
      @@ Parameter:
      @@ R0: Pointer to the ini start address
      @@ R1: Pointer to the section base address
      @@ R2: default data start address
      @@
      @@ R7: used as a decompress function pointer

      .text

      .global inixsc_init
 inixsc_init:

 init_loop:
      STMFD    SP!, {R4, R5, R6, R7, LR}
      MOV  R7, #0
      @ align the init data base address
      ADD  R0, R0, #3
      MOV  R0, R0, LSR #2
      MOV  R0, R0, LSL #2

      @ load the kind filed
      LDR  R3, [R0], #4
      MOV  R12, R3, LSR #24
      SUB  R3, R12, LSL #24
      CMP  R12, #0xff

      @ end record found, so exit
      MOVEQ    R0, #0
      LDMEQIA  SP!, {R4, R5, R6, R7, PC}

      CMP  R12, #11
      MVNHI    R0, #0
      LDMHIIA  SP!, {R4, R5, R6, R7, PC}   @ return error code

      LDR  R12, [PC, R12, LSL #2]
      ADD  PC, PC, R12

 CaseTable:
      .word Case00 - CaseTable
      .word Case01 - CaseTable
```

```
         .word Case02 - CaseTable
         .word Case03 - CaseTable
         .word Case04 - CaseTable
         .word Case05 - CaseTable
         .word Case06 - CaseTable
         .word Case07 - CaseTable
         .word Case08 - CaseTable
         .word Case09 - CaseTable
         .word Case0a - CaseTable


    Case00:
    @ R3: number of bytes which has to be copied
         @ R12: Destination Pointer
         @ R0: Source Pointer
         @ R2: Destination Base Address
         @ R4: Byte
         LDR  R12, [R0], #4
         ADD  R12, R2

         CMP  F7, #0
         BEQ  Case00_loop

         @ use decompress byte stream
         STMFD    SP!, {R1-R3, R12} @ save used register
         MOV  R1, R12  @ load destination pointer
         MOV  R2, R3   @ size of the compressed area
         BLX  R7       @ call decompress function
         LDMIA    SP!, {R1-R3, R12} @ restore register values
         B    init_loop

    Case00_loop:
         SUBS R3, R3, #1
         LDRB R4, [R0], #1
         STRB R4, [R12], #1
         BNE  Case00_loop;
         B    init_loop

    Case01:   @ R3: number of bytes that have to be copied
         @ R12: Destination Pointer
         @ R0: Source Pointer
         @ R2: Destination Base Address
         @ R4: Byte
         LDR  R12, [R0], #4
         ADD  R12, R2

    Case01_loop:
         SUBS R3, R3, #1
         LDR  R4, [R0], #4
         STR  R4, [R12], #4
         BNE  Case01_loop;
         B    init_loop

    Case02:   @ R3: number of bytes that have to be set to a specific value
         @ R12: Destination Pointer
         @ R0: Source Pointer
         @ R2: Destination Base Address
         @ R4: Byte value
```

```
        LDR  R12, [R0], #4
        ADD  R12, R2
        LDR  R4, [R0], #4

    Case02_loop:
        SUBS R3, R3, #1
        STRB R4, [R12], #1
        BNE  Case02_loop;
        B    init_loop

    Case03:   @ R3: number of bytes that have to be set to a specific value
        @ R12: Destination Pointer
        @ R0: Source Pointer
        @ R2: Destination Base Address
        @ R4: long value
        LDR  R12, [R0], #4
        ADD  R12, R2
        LDR  R4, [R0], #4

    Case03_loop:
        SUBS R3, R3, #1
        STR  R4, [R12], #4
        BNE  Case03_loop;
        B    init_loop

    Case04: @ R3: new section index
        @ load R2 with the next section index
        CMP  R1, #0
        MVNEQ    R0, #0
        LDMEQIA  SP!, {R4, R5, R6, PC}   @ return ERROR

        CMP  R3, #255
        MOVEQ    R2, #0
        LDRNE    R2, [R1, R3, LSL #2]
        B    init_loop

    Case05: @ fill in the relocations
        @ R0: init section base address
        @ R1: section base address array
        @ R2: destination base address
        @ R3: Number of relocations
        @ R4: position of the relocation
        @ R5: load register for the relocation
        @ R6: Section base address
        @ R12: calculation register for the relocation
        MOV  R6, R3, LSR #16
        SUB  R3, R3, R6, LSL #16 @ now, R3 contains the number of relocs

        CMP  R6, #255
        MOVEQ    R6, #0
        LDR  R6, [R1, R6, LSL #2] @ R6 contains the What address

    Case05_loop:
        LDR  R4, [R0], #4
        ADD  R4, R4, R2    @ R4 contains the Where address
        TST  R4, #3        @ check the lower two bytes
        LDREQ    R12, [R4] @ Load the 32-bit value
        ADDEQ    R12, R12, R6
        STREQ    R12, [R4] @ STORE the 32-bit value
        BEQ  Case05_Skip

        LDRB R12, [R4, #0] @ load the first byte
        LDRB R5,  [R4, #1] @ load the second byte
        ORR  R12, R12, R5, LSL #8
        LDRB R5,  [R4, #2] @ load the third byte
        ORR  R12, R12, R5, LSL #16
        LDRB R5,  [R4, #3] @ load the fourth byte
        ORR  R12, R12, R5, LSL #24

        ADD  R12, R12, R6

        STRB R12, [R4, #0] @ store the first byte
        MOV  R12, R12, LSR #8
        STRB R5,  [R4, #1] @ store the second byte
        MOV  R12, R12, LSR #8
        STRB R5,  [R4, #2] @ store the third byte
        MOV  R12, R12, LSR #8
        STRB R5,  [R4, #3] @ store the fourth byte

    Case05_Skip:
        SUBS R3, R3, #1
        BNE  Case05_loop

    Case06:
    Case07:
    Case08:
    Case09:
        MVN  R0, #0
        LDMIA    SP!, {R4, R5, R6, PC}   @ return ERROR

    Case0a:
        @ load decompress start address

        CMP  R3, #255
        MVNEQ    R12, #0
        LDRNE    R12, [R1, R3, LSL #2]
        LDR  R7, [R0], #4
        ADD  R7, R7, R12
        B    init_loop

        .END
```

Technical Support

Registration

To receive technical support for this product and product updates, you need to be registered for an Intel® Premier Support account on our secure web site, https://premier.intel.com/. If not yet done, please register your product at https://registrationcenter.intel.com**.**

If you cannot register your product or you cannot access your account, please contact https://registrationcenter.

intel.com/support/contact.aspx and submit your problem.

Note: If your distributor provides technical support for this product, please contact them for support rather than Intel.

Startup Support

For initial startup support such as installation, licensing, support account issues, please visit https://registrationcenter.intel.com/support/contact.aspx.

Product Support

If you need help or if have problems with the product, submit your issues via the Intel® Premier Support at https://premier.intel.com.

Steps to submit an issue:

1. Go to https://premier.intel.com/.
2. Type in your Login and Password. Both are case-sensitive.
3. Click the "`Submit`" button.
4. Read the Confidentiality Statement and click the "`I Accept`" button.
5. Click on the "`Go`" button next to the "`Product`" drop-down list.
6. Click on the "`Submit Issue`" link in the left navigation bar.
7. Choose "`Development Environment (tools,SDV,EAP)`" from the "`Product Type`" drop-down list.
8. If this is a software or license-related issue, choose "`Intel(R) PCA SW Dev Tool Suite, Pro`" from the "`Product Name`" drop-down list.
9. Enter your question and complete the fields in the windows that follow to successfully submit the issue.

Guidelines for problem report:

1. Describe your difficulty or suggestion.
   For problem reports please be as specific as possible, so that we may reproduce the problem. For compiler problem reports, please include the compiler options and a small test case if possible.
2. Describe your system configuration information.
   Refer to `<install_dir>\Doc\support.txt` and provide the package ID listed. Please include any other specific information that may be relevant to helping us to reproduce and address your concern.
3. If you were not able to install the compiler or cannot get the Package ID, enter the filename you downloaded as the package ID.

Additional Information

Related Products and Services

Information on Intel software development products is available at http://www.intel.com/software/products.

Some of the related products include:

- The Intel® Software College provides training for developers on leading-edge software development technologies. Training consists of online and instructor-led courses covering all Intel architectures, platforms, tools, and technologies.
- The VTune™ Performance Analyzer enables you to evaluate how your application is utilizing the CPU and helps you determine if there are modifications you can make to improve your application's performance.
- The Intel® C++ and Fortran Compilers are an important part of making software run at top speeds with full support for the latest Intel IA-32 and Itanium® processors.
- The Intel® Performance Library Suite provides a set of routines optimized for various Intel processors. The Intel® Math Kernel Library, which provides developers of scientific and engineering software with a set of linear algebra, fast Fourier transforms and vector math functions optimized for the latest Intel Pentium® and Intel Itanium processors. The Intel® Integrated Performance Primitives consists of cross-platform tools to build high performance software for several Intel architectures and several operating systems.

Disclaimer and Legal Information