XA Toolchain v4.0r1

RELEASE NOTE

SUMMARY

This release note describes the changes and new features of all TASKING XA products with respect to v3.0r5.

The main reasons for this release are:

EDE

The XA Embedded Development Environment (EDE) has been significantly renewed and extended with the following new features:

- **Project Spaces** - This groups a number of projects into logical units called project spaces, improved project management using right mouse-click menus and clear project definition windows to make adding and removing files to and from a project easy.
- **CodeSense** - To guide and assist you when you write your source code.
- **Improved Tags** - To get a structured overview of your sources and their relations.
- **Snippets** - For easy interactive clipboard kind of behavior.
- **Support for Expert mode.** - For showing 'advanced' options.
- **Application extensions** - Such as an HTML browser, an FTP client and much more that can be easily loaded.

Projects & Project Spaces

All your projects can now be grouped together in Project Spaces. The `XA Examples' project space is the default Project Space containing all XA examples of the product you have installed. If you want to open an other than the default example (project) within this Project Space, just select one, click on the right mouse button and select the `Set as Current Project' menu entry. The EDE make and rebuild commands which you use to build or rebuild your project only work on the current project.Adding files to and removing files from a project has become easier than ever. Just click with your right mouse button on a source file and the pop-up dialog contains an entry to add this file to the current project. Click with your right mouse button on a file in the project window and the pop-dialog contains an entry to remove this file from the project.You can create your own project spaces from the `Project' menu. From this menu you can also add new or existing projects to your own Project Space.

For more information, please refer to the XA EDE online help system.

CodeSense

CodeSense virtually looks over your shoulder and gives you useful information in the form of hints as you type your source code. For example if you are programming a `printf` statement, it will show you the next expected parameter and the prototype of this function in a small yellow balloon-help box. If you have already defined a structure with numerous members and from a certain location within your code and you want to access a member, just type in the structure name and a dot, and CodeSense will show you a list with all possible members. You can select a member from this list or search for where this member is defined. This also works for C++ language elements. If you hover the mouse pointer over a function name, CodeSense will show the prototype of this function. This also applies to variables, structures, etc. Just hover the mouse pointer over a C or C++ language element and CodeSense will show you whatever information is relevant.

In order to have all this information ready at hand, CodeSense automatically builds a database

using all the files in the "`include`", "`include.cpp`" and "`examples`" directories when you first start the XA EDE. This is shown by the CodeSense green light which is displayed near the right bottom corner of the EDE window which will become gray when this database is completed. That is the signal that CodeSense is operational.

You can also add your own databases by clicking with the left mouse button on the CodeSense light and select the libraries option. Please follow the instructions to add your own CodeSense information from your own application to the CodeSense database.

Browsing Tags

Tags reflect the cross references in your application. By building a Tags file and graphically browsing your source code using this Tags file, you can get a good overview on cross references in your application such as, which global variables are defined and where they are used, which enumeration types are defined, which global functions are defined and where are they used. Browsing of Tags can help you in getting to know someone else's source code easily and quickly without the need to dive into every detail. From the `Projects' menu select the `Build Tags' option to build a Tags file which reflects your current project. Next, open this Tags file from the browse window which you find in the output window in the bottom of the EDE window. Please select the `Browse' TAB in this window and open the Tags file which has the same name as your current project using the "`*.ptg`" file extension. Start browsing your application.

Snippets

The XA EDE comes with some pre-build Snippets which can aid you in improving your coding speed and efficiency. Basically, Snippets are cut-copy-paste pieces of text which you can select from the Snippets library and drop into your source code. The pre-built Snippets are available from the Snippets library that can be accessed by selecting the right most icon (the CodeFolio button) which is located under the left pane project window. You can even create your own Snippets by simply copying a piece of text or source code to the clipboard, click with the right mouse button on the Snippets library and select to add your Snippet to it. Snippets can also contain some interactive elements which will be activated when you drop a Snippet into your source code. For example, if you select a function header Snippet, it will ask for the name of the function which will automatically be filled in when the Snippet is dropped into the source code. Please see the XA EDE online help system for more information.

Support for Expert mode

For the average project a lot of options offered in the EDE are unnecessary. To hide these 'advanced' options EDE has been extended with an extra menu option 'Expert Mode'. Selecting this mode (shown by the menu entry being checked) will result in the dialogs showing all available options, just like the default situation in previous versions. Deselecting this entry (unchecked menu entry) will result in a much simpler interface. Only the most common options will be shown in that case.

Application extensions

There are a number of XA EDE application extensions such as, an HTML browser and an FTP client which are not available by default. These applications must be loaded in the EDE before

they can be used. Please select the `Tools' menu, select the `Customize' menu entry and finally select the `Libraries' sub-menu entry. This will open a list of application extensions which can be loaded by selecting the extensions you want to use.

C COMPILER

Flexible License Manager for Windows

The XA software is now protected by the FLEXlm license management software. Carefully read the *Software Installation* chapter and the *Flexible License Manager (FLEXlm)* appendix in the XA C Cross-Compiler User's Guide for detailed information.

SmartXA Support

This release of the TASKING XA toolchain supports the NXP SmartXA. This includes SmartXA-specific Special Function Register Support, SmartXA EEPROM, SmartXA memory mappings, SmartXA External memory access protection and SmartXA simulator.

SmartXA-specific Special Function Register Support

The SmartXA SFR file `regsmart.sfr` is part of the XA product and located in the `include` directory. The SmartXA CPU derivative is integrated in EDE (Embedded Development Environment) under the `Processor Options` tab. Selecting it will automatically setup the EDE for SmartXA support. This includes automatic SmartXA SFR file selection, on-chip memory configuration and debugger SmartXA CPU selection.

For CrossView Pro support of the SmartXA registers the SFR definition file `regsmart.def` has been included. This file is located in the `etc` directory.

For non-EDE environments the SmartXA CPU selection is supported via the command line option **-Csmart** for the the control program, compiler, assembler and debugger.

SmartXA CPU and Memory description

The SmartXA CPU and memory description support is part of the XA product and delivered as part of the SmartXA locator description files `smart.cpu`, `smart.dsc`, `smart_t.dsc`, `smart_s.dsc`, `smart_m.dsc`, `smart_c.dsc`, `smart_l.dsc`, `smart_t.i`, `smart_s.i`, `smart_m.i`, `smart_c.i` and `smart_l.i` located in the `etc` directory.

The locator description files are automatically selected by EDE when the SmartXA CPU is selected. For non-EDE environments the locator description files are automatically selected by the control program **ccxa** via the command line option **-Csmart**.

For configuration and control of the SmartXA address space a number of locator controls have been added, which are part of the SmartXA locator description files. These locator controls are under control of the EDE environments. For non-EDE environments these locator controls can be used for direct locator invocation on command line or in invocation files.

The following new locator controls are supported by EDE:

After selection of the SmartXA CPU via the tab `Processor Options | Processor` the `_REGSFR` locator control is configured for SmartXA.

After SmartXA CPU selection internal RAM (`_IRAM`) and ROM (`_ICODE`) size are configurable via the EDE tab `Processor Options | Memory`. On-chip EEPROM/ROM program size selection conforms the Page Zero or Large memory mode, which depends on the memory model selection. Memory model selection is available via the EDE tab `C Compiler Options | Project Options | Memory model`. The `_FAME` and `_EEPROM` controls do not need any EDE configuration after SmartXA CPU selection.

The locator control `_SM` is configured with startup selection via the EDE tab `Processor Options | PSW`. Selection of `System mode` radio button on this tab will define the `_SM` control.

The order and absolute address of EEPROM data sections can be defined via the EDE tab `Linker/Locator Options | Data`, which control the `_EORDER` and `_DATA` locator controls.

To reserve a memory area in the EEPROM data space you can define an address range via the EDE tab `Linker/Locator Options | Reserve` in the edit field `Reserve Data EEPROM area(s)`. The locator controls `_ERESERVES` and `_ERESERVED` are used by EDE for this configuration.

For the configuration of RAM segmentation you can define the DMCR bits via the EDE tab `Processor Options | DMCR`. Not only DMCR startup values are configured but also the RAM data sapce for locating is configured via the Locator controls `_SSS` and `_USS`.

For the configuration of the window length you can define the `MUBLKHI0` and `MUBLKHI1` MMU registers via the EDE tab `Processor Options | MMU`. Not only MMU startup values are configured but also the window size for locating User Mode applications is configured via the locator controls `_MUBLKHI0` and `_MUBLKHI1`.

An overview of the SmartXA specific locator controls is given in the following table:

Locator control	**Default**	**Description**
Compiler and derivative used:		
_REGSFR	regsmart.dat	Special Function Register file for CrossView Pro Debugger
Processor configuration:		
_IRAM	0A20H	Internal RAM size
_SSS	_IRAM-_FAME	Internal RAM in segment 0
_USS	128	Internal RAM in segment 1..N
_FAME	544	Internal RAM in segment 15
_ICODE	32k	Internal ROM size
_EEPROM	32k-64	Internal EEPROM size
_MUBLKHI0	255	Internal ROM window 0 (in 256 by te blocks) 0..255
_MUBLKHI1	255	Internal ROM window 1 (in 256 by te blocks) 0..255
_SM	1	0: User mode 1: System mode
Internal memory:		

_ERESERVES	False	Define if you want to reserve some memory area on the EEPROM bus
_ERESERVED(*startaddr*,*endaddr*)	--	Reserve memory area on the EEPROM bus: start-address, end-address
Order (and optional address) of user sections:		
_EORDER	False	Define if you want to use the _EDATA control
_EDATA(*name* [addr=*addr*])	--	Specify section with *name* must be located at address *addr* or before/after another _EDATA control. Must be within #ifdef _EDATA and #endif

SmartXA EEPROM Support

The SmartXA is equipped with a 32Kbytes EEPROM. This non-volatile memory is available through data and code space. The SmartXA allows EEPROM data access in System Mode only.

The EEPROM ROM data access is supported by the C compiler via the `_rom` storage type qualifier. The EEPROM RAM data access is supported by the C-compiler via the `_edata` storage type qualifier. This qualifier allows C programming without using the EEPROM SFRs; the C compiler takes care of generating the EEPROM SFRs for accessing the EEPROM data memory.

The EEPROM is non-volatile RAM. Variables located in non-volatile RAM are retained when the CPU is reset or turned off. Therefore, the `_edata` storage qualifier implies for global data objects not being cleared or initialized at startup. The ANSI-C standard prescribes that 'normal' not initialized non-automatic variables are cleared at startup and initialized, which is unwanted for non-volatile RAM like the EEPROM.

A new space has been added to the assembler called EDATA space to support EEPROM access as data memory. The EDATA storage qualifier implies for global data objects not being cleared or initialized at startup. The absolute section directive ESEG and the the relocatable section attribute EDATA are added for this purpose. The EDATA storage support is only available if the SmartXA CPU is selected from the EDE environment or when **-Csmart** is defined as command line option for the assembler.

SmartXA External memory access protection

For the SmartXA there is no access mechanism for external memory. The assembler does not support external data space if the SmartXA CPU is selected from the EDE environment or when **-Csmart** is defined as command line option for the assembler.

XA-SCC, XA-H3, XA-H4 Support

This release of the TASKING XA toolchain supports the N XA-SCC. This includes SCC-specific Special Function Register Support and an Extra Default Data Segment via the Compact memory model.

This release also supports the XA-H3 and XA-H4 derivatives.

SCC-specific Special Function Register Support

The XA-SCC SFR file `regxascc.sfr` is part of the XA product and located in the `include` directory. The

XA-SCC CPU derivative is integrated in EDE (Embedded Development Environment) under the `Processor Options` tab. Selecting it will automatically setup the EDE for XA-SCC support. This includes automatic XA-SCC SFR file selection, on-chip memory configuration, Unified External bus and debugger XA-SCC CPU selection.

For CrossView Pro support of the XA-SCC registers the SFR definition file `regxascc.def` has been included. This file is located in the `etc` directory.

For non-EDE environments the XA-SCC CPU selection is supported via the command line option **-Cxascc** for the compiler, assembler and debugger.

H3/H4-specific Special Function Register Support

As with the SCC for the H3 and H4 the SFR files `regxah3.sfr` and `regxah4.sfr` are located in the `include` directory.

For CrossView Pro support of the XA-H3/H4 registers the SFR definition files `regxah3.def` and `regxah4.def` have been included. These files are located in the `etc` directory.

For non-EDE environments the XA-H3/H4 CPU selection is supported via the command line option **-Cxah3** and **-Cxah4** respectively.

Relocatable Memory Mapped Registers

The XA-SCC Memory Mapped Registers (MMRs) are part of the XA product and delivered as the compiler include file `cxascc.h` and the assembler include file `cxascc.inc` located in the `include` directory. These files are NOT automatically included by the compiler or assembler and are NOT part of automated EDE tool control. Usage is supported by explicitly using the include files in the XA-SCC application source code.

For the XA-H3 and H4 the compiler include file `cxah3h4.h` and the assembler include file `cxah3h4.inc` are present.

Extended Memory description for the XA-SCC

The XA-SCC external unified bus support is part of the XA product and delivered as part of the standard locator description file `xa.mem` located in the `etc` directory. The external unified bus is supported by EDE in the tab `EDE | Linker/Locator Options... | Memory Unified`.

For non-EDE environments the locator macro `_UNIFIED_BUS` is added to control the unified bus support for the standard locator descriptions.

Generic Pointer Support

The TASKING XA toolchain has been extended with generic pointer support.

The TASKING XA C compiler allows you to define generic or universal pointers with the `_generic` keyword. When you define a generic pointer it does not point to a specific address space. Instead the address space information is contained in the generic pointer and evaluated at run-time.

Huge memory model

The TASKING XA toolchain has been extended with an extra memory model 'huge'.

The huge memory model supports huge as the default storage type for unqualified data. Objects in the huge memory model can reside anywhere in the 16M data space and the object size is not restricted to 64Kb. Generic pointers are used for unqualified pointers in the huge model. A default pointer can point to the code or data space in the huge memory model. The advantage of this model is that you never need to qualify storage for accessing the whole XA code and data space. The disadvantage is a large code size and code speed draw back.

Huge memory allocation

The C libraries are extended with huge memory allocation functions for the Medium, Compact and Large memory model. These new C library functions are called, halloc, hfree, hcalloc and hrealloc. In contrast with the ANSI-C compliant memory allocation functions (malloc, calloc, realloc and free), these huge memory allocation functions have no 64Kb block or boundary restrictions.

Extra default data segment in Compact memory model

The TASKING XA toolchain is extended with an extra default data segment in the XA Compact memory model.

This new Compact memory model is supported by the EDE via tab `EDE | C Compiler Options | Memory Model | Compact`. The Compact memory model is also supported via the compiler command line option **-Mc**. New C libraries are added for the Compact memory model. These new C libraries are `libcc.a`, `libccs.a`, `libfpc.a` and `libfpct.a`. For locating, two new default description files are added, called `xa_c.dsc` and `xa_c.i` which are located in `cxa/etc`. The XA control program **ccxa** is updated to support recognition and passing Compact memory model selection. Include files and startup code is updated to be aware of the Compact memory model.

The Compact memory model supports an extra default data segment, accessed via Extra segment register ES. The storage space qualifier `_far_es` is added to the C compiler to support this extra default data segment.

A register protocol is implemented for accessing this extra default data segment. The registers R4 and R5 are used for `_far_es` data access. From the startup code the segment selection register SSEL is initialized at `30H` and should remain this value. All data access changing SSEL must restore its default value.

Code Segment Register Based ROM Data Access

The TASKING XA toolchain is extended with Code Segment register based ROM data access for the Compact memory model.

The default ROM data segment is supported by the C compiler via the `_rom_cs` storage type qualifier. The default ROM data segment is accessed via segment register CS.

The advantage of the `_rom_cs` storage qualifier in the Compact memory model is the extra 64K of 16-bit indirect addressable memory in default ROM data segment CS. The disadvantage of `_rom_cs` support in the Compact memory model is the extra overhead when using `_rom`, because the compiler needs to set CS back to its original value after each 24-bit ROM data access.

A register protocol is used for accessing the default ROM data segment CS. The register protocol conforms to

the protocol already available in the Compact memory model for default RAM data access via Extra segment register ES. The registers R4 and R5 are used for `_rom_cs` data access. Therefore, the Segment Selection register SSEL must be initialized at startup or by an operating system with 030H and should remain this value.

Program Counter Based ROM Data Access

The TASKING XA toolchain is extended with indirect ROM data access in the segment identified by the program counter for the Compact and Medium Memory model. This ROM data access is called Program counter based ROM data access.

The Program counter based ROM data access is supported by the C compiler via the `_rom_pc0` and `_rom_pc1` storage type qualifiers. The storage type specifiers `_rom_pc0` and `_rom_pc1` specify an object that is located in one of the two PC ROM data segments. These PC ROM data segments will allow for an additional space of 2*64K bytes of 16-bit accessible ROM data in a 16M code space.

ROM data access does not require any loading of segment registers when ROM data object and function accessing it are both qualified with the storage type specifier `_rom_pc0` or `_rom_pc1`. Two ROM data groups are available for indirect ROM data access via the segment identified by PC. These two ROM data groups, called PC0 and PC1, can reside in any code segment of the XA.

To support Program Counter based ROM data access a register protocol is used which supports default ROM data access via PC. This protocol can be supported in the Medium memory model, SSEL is 0, which allows default indirect ROM data access via register R0 until R6 extended with the 8-bit segment identifier of PC. Also the Compact memory model can support this register protocol efficiently. SSEL is 0x30 by convention that allows default indirect ROM data access via register R0 until R3 and R6 extended with the 8-bit segment identifier of PC.

Allocation of data group in specified segment

Support is added to join a number of data segments into a named data group. The segment attribute **JOIN** is added to the assembler and a **group** directive is added to the DELFEE language. For the compiler default data groups DS and ES support is added to EDE, to allocate these default groups in a user specified segment. Which segment numbers can be assigned to default data groups depends on the processor mode System/User and on the memory model used. Data group allocation is supported by EDE via the tab `Linker/Locator Options | Group`.

Locator control	**Default**	**Description**
Group data sections:		
_DGROUP	False	Define if you want to use the _DGROUPNR control
_DGROUPNR(*name*, *number*)	--	Specified RAM data group with *name* must be alllocated in page *number*. Must be within #ifdef _DGROUP and #endif
_RGROUP	False	Define if you want to use the _RGROUPNR control
_RGROUPNR(*name*, *number*)	--	Specified ROM data group with *name* must be alllocated in page *number*. Must be within #ifdef _RGROUP and #endif
_CGROUP	False	Define if you want to use the _CGROUPNR control
_CGROUPNR(*name*, *number*)	--	Specified ROM code group with *name* must be alllocated in page *number*. Must be within #ifdef _CGROUP and #endif

_STACK_GP	DS	Define if you want to group the stack to a (default) data group
_HEAP_GP	DS	Define if you want to group the heap to a (default) data group

Application Mode Selection

The TASKING XA C compiler has been extended with support for optimal CPU mode code generation and control.

The compiler option **-m***cpumode* has been added to define the application CPU mode to be used. The application CPU mode code generation can be System mode only (**-ms**), User mode only (**-mu**) or Mix mode (**-mm**). This new compiler option is supported by EDE via the tab `C Compiler Options | Project Options | Memory Model`.

For optimal code generation it is required to define if an application runs in System mode or User mode. Code generation for Mix mode supports both execution modes, but increases code size due to run-time checks. This option only affects the code generation for 24-bit stack access in the Large Memory model. The Large memory model libraries are compiled for Mix mode.

This option also controls the availability of compiler features in all memory models. For example, interrupt functions are not allowed in user mode.

Startup code configuration

All processor configurations done by the startup code can be configured. The startup configurations involved are System Configuration Register, Reset Program Status Word and Initializations. These startup configurations can be made within EDE via the tab `Processor Options` after the startup code has been added to your project. These startup configurations are also supported via the following assembler macro definitions, which can be used in non-EDE environments.

Define	**Description**
Initialization	
__NOINITSEG	If defined, C variables are not initialized at startup
__NOINITBIT	If defined, the bit-addressable range is not cleared at startup. (default not defined)
__NOINITARG	If defined, disable initilization of argc and argv
__NOAVOIDNULL	If defined, no section is defined to prevent pointers to be allocated at address 0 (NULL). (default not defined)
__NOEXIT	If defined, C exit() and abort() are not supported in startup
__NOWATCHDOG	If defined, disable watch dog code
__ESWEN	If defined, write through ES in User mode is enabled
Configuration bits SCR	
__PT1 __PT0	Peripheral Clock coreXA SmartXA 0 0 oscillator/4 oscillator/4 (default) 0 1 oscillator/16 oscillator/3 1 0 oscillator/64 oscillator/2

	1 1 reserved oscillator/1
__CM	Defined to 0, because only "native" mode XA is supported
__PZ	1 for Page zero mode (default) 0 for Large memory mode
__EEFAST	1 EEPROM fast timing is active (SmartXA only) 0 EEPROM fast timing is inactive (default)
Configuration bits Reset PSW	
__SM	1 for System mode (default) 0 for User mode
__TM	1 for Trace Mode 0 for XA debugging feature are disabled (default)
__RS	Register bank number [0..3] (default 0)
__IM	Interrupt priority (0-15), 15 indicates highest priority (default)
__C	Carry flag (default 0)
__AC	Auxiliary carry (default 0)
__V	Overflow flag (default 0)
__N	Negative flag (default 0)
__Z	Zero flag (default 0)
Configuration bits DMCR (SmartXA only)	
__SSS1 __SSS0	Size of segment 0 (bytes) 0 0 256 0 1 512 1 0 1024 1 1 Maximum
__USS1 __USS0	Size of segment 1-14 (bytes) 0 0 128 0 1 256 1 0 512 1 1 Maximum
Configuration Memory Management Unit (SmartXA only)	
__MUBLKHI0	Highest block number window 0 (default 255)
__MUBLKHI1	Highest block number window 1 (default 255)
__MUBAS0	Relocation offset window 0 (default 0)
__MUBAS1	Relocation offset window 1 (default 0)

The default SmartXA startup libraries `libsc.a`, `libsl.a`, `libsm.a`, `libss.a` and `libst.a` are added to the `lib\xa` directory. The SmartXA startup libraries contain default startup code for the SmartXA. These libraries only contain `start.obj` for each memory model. These libraries are automatically linked if you select the SmartXA CPU from the EDE environment or when you specify **-Csmart** as a command line option to the control program **ccxa**. When the **lkxa** linker is used without control program, specify these libraries in

front of all other libraries with **-ls***model*. For example, with **-lsm** the linker is looking for the medium SmartXA startup library `libsm.a`.

File System Simulation / C Library Updates

The C library has been updated for a beter file I/O support using File System Simulation (FSS). The following header files are new: `fcntl.h`, `fss.h` and `unistd.h`. The interface of the low level input/output routines `_ioread()` and `_iowrite()` have been changed. A file handle is now passed to these functions instead of a file pointer. The prototypes of `_ioread()` and `_iowrite()` are removed from the system include file `stdio.h`. The low level input/ouput routines `_read()` and `_write()` are now the interface to the standard C input/output functions.

Exclusive Stack Parameter Passing

The TASKING XA C compiler has been extended with the function qualifier `_stackparm`. The `_stackparm` function qualifier can be used for functions to define that all arguments are passed via the stack. The keyword `_stackparm` is allowed with function declarations and function prototypes.

Interrupt vector offset

To support applications using ROM monitors the interrupt vector base address can be specified for code generation. Vector table entries generated by the compiler are located at the vector address incremented with a user specified interrupt vector offset. From EDE the Interrupt vector offset can be specified via tab `EDE | C compiler Options | Allocation | Interrupt vectors`. The interrupt vector offset is also supported via the compiler command line option **-ivo=***interrupt_vector_offset_address*.

Interrupt vector list

The functions qualifiers `_interrupt()` and `_using()` for interrupt service routines are extended to accept a list of arguments. This extension makes it possible to bound more than one interrupt vector to one interrupt service routine. It is very useful when you want to bound all non-used interrupt vectors to one function which executes a processor trap instruction.

Interrupt Function Qualifiers

The TASKING XA C compiler has been extended with the interrupt function qualifiers `_pagezero` and `_frame`.

You can use the `_pagezero` function qualifier for `_interrupt` functions to define that the interrupt function itself is located in segment zero. The jump chain required in Large memory mode for calling interrupt functions outside segment zero is suppressed for `_pagezero` qualified interrupt service routines. This decreases the interrupt latency for the Large memory mode models Medium, Compact and Large.

With the `_frame` function qualifier you can specify which registers must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack. The syntax is:

_frame(*reg*[**,***reg*]... **)**

where, *reg* can be one of the following registers: R0..R6,DS,CS,ES or SSEL.

A warning is generated if some registers are missing which are normally required to be pushed and popped in an interrupt function prolog and epilog to avoid run-time problems.

Example:

```
_interrupt(1) _using(0x8f00) _frame(R0,R1)

void alarm( void )

{

  /* an interrupt function */

}
```

Function Call with Software Trap Instruction

The TASKING XA C compiler has been extended with the function qualifier `_trap`. The `_trap` function qualifier can be used for functions to define that a function is called via a software trap instruction. The keyword `_trap` is allowed with function declarations and function prototypes. Contrary to an interrupt function a function declared with `_trap` can have parameters and can have a return value.

The syntax is:

_trap(*trap_nr* **) _using(** *psw* **)**

The `_trap` function qualifier takes one argument *trap_nr* that defines the software trap number. *trap_nr* is a number in the range 0-15.

The `_using` function qualifier must be used in combination with the `_trap` function qualifier to define the value of PSW placed in the interrupt vector table.

The *trap_nr* is filled by the compiler (unless disabled by the **-v** option or **novector** pragma) with the software trap number and using number specified. The software trap vector range is -1 or 0-15.

SFR Include Files

The Special Function Register include files are now protected against double inclusion. All SFR files are added with a `_REGXA`*`cpu`*`_SFR` define, which corresponds to the processor name. You can also use these defines in C source to determine which processor derivative is used.

Intrinsic Functions

The following intrinsic functions have been added in this release:

Function	Description
_getsp()	Get stack pointer value
_getusp()	Get user stack pointer value
_setsp()	Set stack pointer value

_setusp()	Set user stack pointer value
_addsp()	Add value to stack pointer
_offsp()	Get stack offset

See the *XA C Cross-Compiler User's Guide* for a description.

char type bit field and enumeration optimizations

To optimize allocation of enumerations and bit fields the C compiler can treats bit fields and enumerations that fit in 8-bit as 'char' types instead of 'int' types. Also bit fields can now be declared as 'char' types instead of 'int' types. These data allocations can be enabled with the **-Tce** compiler command line option or the EDE storage allocation options. See EDE | C Compiler Options | Project Options... menu item and select the Allocation tab.

char type structure packing

Structures and nested structures that only contain character type members can be packed. A command line option **-Tp** and pragma's **packchar** and **endpackchar** are added to the C compiler to control packing of char type structures. See EDE | C Compiler Options | Project Options... menu item and select the Allocation tab.

MISRA C

MISRA C consist of a large set of rules that can be used to restrict the use of dangerous and obscure C constructions. The compiler supports automatic checking of your code against these rules. Each of the MISRA C rules can be switched on and off separately using a -safer <rule number> option. EDE is extended with a separate menu allowing each of the rules to be switched on or off. Also it allows for selection of the MISRA C guidelines (see http://www.misra.org.uk).

ASSEMBLER

Assembler PC relative

A new optimization option has been added to the assembler. The command line option **-Of/-OF** enables or disables conversion of generic instructions to far instructions on out-of-range detection. This option is only available for Large memory mode.

These new assembler option are supported by the EDE via the tab `Assembler Options | Optimization`.

Section Summary

The **-t** option of the assembler can optionally have flags **c** and **l**. **-tc** displays a section summary on `stdout`. **-tl** displays a section summary in the list file. **-tcl**, which is the same as **-t**, does both.

UTILITIES

Utility rmxa removed

The register manager utility has been removed from the TASKING XA toolchain, because it is no longer needed to generate register number mappings for CrossView Pro. Instead CrossView Pro uses the special function register files stored in the `cxa\include` directory.

CROSSVIEW PRO

SmartXA Support

This release of the TASKING XA toolchain supports the X SmartXA. This includes SmartXA-specific Special Function Register Support, SmartXA EEPROM, SmartXA memory mappings, SmartXA External memory access protection and SmartXA simulator.

SmartXA-specific Special Function Register Support

For CrossView Pro support of the SmartXA registers the SFR definition file `regsmart.def` has been included. This file is located in the `etc` directory.

For non-EDE environments the SmartXA CPU selection is supported via the command line option **-Csmart** for the the control program, compiler, assembler and debugger.

XA-SCC Support

This release of the TASKING XA toolchain supports the NXP XA-SCC. This includes SCC-specific Special Function Register Support and an Extra Default Data Segment via the Compact memory model.

SCC-specific Special Function Register Support

For CrossView Pro support of the XA-SCC registers the SFR definition file `regxascc.def` has been included. This file is located in the `etc` directory.

For non-EDE environments the XA-SCC CPU selection is supported via the command line option **-Cxascc** for the compiler, assembler and debugger.

Simulator Peripheral Support

The simulator has been extended with support for 3 timers. Implementation of these timers is conform the implementation on the real XA-G3. All modes including the enhanced mode 0 have been implemented. Using the timers in the simulator does not require any additional programming. A program using timers that runs on a real processor should run the same way on the simulator. This includes the handling of interrupts on timer overflows.

GUI Enhancements

Several dialogs have been improved. The Communication Setup dialog and the Expression Evaluation dialog have been redesigned. The Select CPU dialog has been added.

Reset Behavior

The reset behavior of CrossView Pro has been changed. A distinction between "program reset" and "target system reset" has been introduced. A "program reset" sets the program counter to the application's start address, while the "target system reset" executes a hardware reset.

Solved problems

A large number of problems has been solved in this release. They are described in separate files: "`solved_<name>.html`".
