AMX®

AMX™ for ARM Target Guide

First Printing: August 1, 1997
Last Printing: November 1, 2007

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5
Phone: (604) 734-2796
Fax: (604) 734-8114

TECHNICAL SUPPORT

KADAK Products Ltd. is committed to technical support for its software products. Our programs are designed to be easily incorporated in your systems and every effort has been made to eliminate errors.

Engineering Change Notices (ECNs) are provided periodically to repair faults or to improve performance. You will automatically receive these updates during the product's initial support period. For technical support beyond the initial period, you must purchase a Technical Support Subscription. Contact KADAK for details. Please keep us informed of the primary user in your company to whom update notices and other pertinent information should be directed.

Should you require direct technical assistance in your use of this KADAK software product, engineering support is available by telephone, fax or e-mail. KADAK reserves the right to charge for technical support services which it deems to be beyond the normal scope of technical support.

We would be pleased to receive your comments and suggestions concerning this product and its documentation. Your feedback helps in the continuing product evolution.

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5

Phone: (604) 734-2796
Fax: (604) 734-8114
e-mail: amxtech@kadak.com

DISCLAIMER

TRADEMARKS

AMX for ARM TARGET GUIDE

Table of Contents

AMX for ARM TARGET GUIDE
Table of Contents (Cont'd)

Appendices

AMX for ARM TARGET GUIDE
Table of Figures

1. Getting Started with AMX for ARM

1.1 Introduction

The AMX™ Multitasking Executive is described in the AMX User's Guide. This target guide describes AMX 4-ARM and AMX 4-Thumb. AMX 4-ARM operates on the ARM Ltd. ARM™ v4, v4T, v5 and v5T processors and all architecturally compatible processors. AMX 4-Thumb operates only on the ARM v4T or v5T processor and architecturally compatible processors.

Throughout this manual, the term ARM refers specifically to the ARM Ltd. family of processors which conform to the ARM v4 or v5 architecture specification. When distinctions are not important, the term ARM is used to reference any processor which has the general characteristics of the ARM v4 or v5 family. When distinctions are important, the processors are identified explicitly.

Note that AMX 4-ARM executes as ARM code on any processor which adheres to the ARM v4 or v5 architecture specification. Consequently, AMX 4-ARM will also operate, albeit strictly as ARM code, on any processor which adheres to the ARM v4T or v5T Thumb architecture specification.

An alternate product, AMX 4-Thumb, is available for developers wishing to use features which are unique to the ARM v4T and v5T Thumb architecture. AMX 4-Thumb supports both Thumb and ARM execution states. Mixed Thumb and ARM applications or Thumb only solutions are supported.

The purpose of this manual is to provide you with the information required to properly configure and implement an AMX 4-ARM or AMX 4-Thumb real-time system. It is assumed that you have read the AMX User's Guide and are familiar with the architecture of the ARM processor.

Installation

AMX 4-ARM and AMX 4-Thumb are delivered ready for development use on a PC or compatible running Microsoft® Windows®. To install AMX, follow the directions in the Installation Guide. All AMX files required for developing an AMX application will be installed on disk in the directory of your choice. All AMX source files will also be installed on your disk.

AMX Tool Guides

This manual describes the use of AMX in a tool set independent fashion. References to specific assemblers, compilers, librarians, linkers, locators and debuggers are purposely omitted. For each tool set with which AMX 4-ARM has been tested by KADAK, a separate chapter in the **AMX 4-ARM Tool Guide** is provided. AMX 4-Thumb users should refer to the **AMX 4-Thumb Tool Guide**.

1.2 AMX Files

AMX is provided in C source format to ensure that regardless of your development environment, your ability to use and support AMX is uninhibited. AMX also includes a small portion programmed in ARM assembly language.

Figures 1.2-1, 2 and 3 summarize the AMX modules provided with AMX 4-ARM. If you are using AMX 4-Thumb, replace references to KADAK part number 402 with the Thumb part number 422. The AMX product manifest (file *MANIFEST.TXT*) is a text file which indicates the current AMX revision level and lists the AMX modules which are provided with the product.

File Name	**Module**
CJ402 .H	Generic include file
CJ402APP.H	Custom application definitions
CJ402CC .H	C dependent definitions
CJ402EC .H	AMX error code definitions
CJ402IF .H	C and target interface prototypes
CJ402KC .H	Private AMX constants
CJ402KF .H	AMX service procedure prototypes
CJ402KP .H	Private AMX prototypes
CJ402KS .H	Private AMX structure definitions
CJ402KT .H	Target processor definitions
CJ402KV .H	AMX version specification
CJ402SD .H	AMX application structure definitions
CJ402TF .H	Target dependent prototypes
CJZZZ .H	Copy of generic include file *CJ402.H* used for portability
CHxxxxx .H	Definitions for common timer (PIT) and serial I/O (UART) chips

Figure 1.2-1 AMX Include Files

File Name	**Module**
CJ402K .DEF	Private AMX assembly language definitions
CJ402KQ .S	Private AMX math procedures
CJ402KR .S	AMX Interrupt Supervisor
CJ402KS .S	AMX Task Scheduler
CJ402MXA.S	Message Exchange Manager constants
CJ402TDC.S	Time/Date Manager constants
CJ402UA .S	Target processor and C support
CJ402UB .S	Target processor and C support

Figure 1.2-2 AMX Assembler Source Files

File Name	**Module**
CJ402KA .C	Kernel task services
CJ402KB .C *CJ402KBR.C*	General task services
CJ402KC .C *CJ402KCR.C*	Timer Manager
CJ402KD .C *CJ402KDR.C*	Task management services
CJ402KE .C	Task termination services
CJ402KF .C	Suspend/resume task
CJ402KG .C	Time slice services
CJ402KH .C	Task status
CJ402KI .C	Enter and Exit AMX
CJ402KJ .C	General object access
CJ402KK .C	AMX Vector Table access
CJ402KL .C	Private AMX list manipulation
CJ402KM .C	AMX task scheduler hook services
CJ402KX .C	AMX Kernel Task
CJ402CL .C	Circular List Manager
CJ402LM .C	Linked List Manager
CJ402BM .C *CJ402BMR.C*	Buffer Manager
CJ402EM .C *CJ402EMR.C*	Event Manager
CJ402RM .C	Semaphore Manager (resources)
CJ402SM .C *CJ402SMR.C*	Semaphore Manager
CJ402MB .C *CJ402MBR.C*	Mailbox Manager
CJ402MF .C	Flush mailbox and message exchange
CJ402MM .C *CJ402MMR.C*	Memory Manager
CJ402MX .C *CJ402MXR.C*	Message Exchange Manager
CJ402TDA.C	Time/Date Manager
CJ402TDB.C	Time/Date formatter
CJ402UF .C	Launch and leave AMX
CJ402XTA.C	Message exchange task services
CJ402XTB.C	Message exchange task termination
CHxxxxxT.C	Clock drivers for common timer (PIT) chips
CHxxxxxS.C	Sample drivers for common serial I/O (UART) chips

Figure 1.2-3 AMX C Source Files

1.3 AMX Nomenclature

The following nomenclature standards have been adopted throughout the AMX Target Guide.

Numbers used in this manual are decimal unless otherwise indicated. Hexadecimal numbers are indicated in the format *0xABCD*.

The terminology *A(Table XYZ)* is used to define addresses. It is read as "the address of Table XYZ".

Read/write memory is referred to as RAM. Read only memory (non-volatile storage) is referred to as ROM.

AMX symbol names and reserved words are identified as follows:

cjkkpppp	AMX C procedure name *pppp* for service of class *kk*
cjxtttt	AMX structure name of type *tttt*
xttttyyy	Member *yyy* of an AMX structure of type *tttt*
CJ_ID	AMX object identifier (handle)
CJ_ERRST	Completion status returned by AMX service procedures
CJ_CCPP	Procedures use C parameter passing conventions
CJ_ssssss	Reserved symbols defined in AMX header files
CJ_ERxxxx	AMX Error Code *xxxx*
CJ_WRxxxx	AMX Warning Code *xxxx*
CJ_FExxxx	AMX Fatal Exit Code *xxxx*
CJ402xxx.xxx	AMX 4-ARM filenames
CJ422xxx.xxx	AMX 4-Thumb filenames
CJZZZ.H	Generic AMX include file

The generic include file *CJZZZ.H* is a copy of the AMX 4-ARM header file *CJ402.H* (or AMX 4-Thumb header file *CJ422.H*) which includes the subset of the AMX header files needed for compilation of your AMX application C code. By including the file *CJZZZ.H* in your source modules, your AMX application becomes readily portable to other target processors.

Throughout this manual code examples are presented in lower case. File names are shown in upper case. C code assumes that an *int* is 32 bits as is common for most C compilers for the ARM processor.

Processor registers are referenced using the software names specified by the ARM Procedure Call Standard (APCS).

```
a1 to a4, v1 to v7, fp, ip, sp, lr, pc, CPSR, SPSR
```

When the context requires, the banked processor registers (*sp*, *lr* and *SPSR*) are referenced using the ARM Ltd. extensions as in *R13_SVC*, *R13_ABORT*, *R13_UNDEF*, *R13_IRQ* and *R13_FIQ*. The extra FIQ banked registers are always referenced by name as in *R8_FIQ* to *R12_FIQ*.

1.4 AMX for ARM Target Specifications

AMX 4-ARM and AMX 4-Thumb were initially developed and tested using the ARM7TDMI core processor on the ARM Development Platform produced by ARM Ltd. However, the AMX 4-ARM and AMX 4-Thumb design criteria fully encompass the ARM v4, v4T, v5 and v5T processor family requirements.

AMX uses a set of design constants which vary according to the constraints imposed by each target processor. When operating on the ARM processor, these design constants assume the values listed in Figure 1.4-1.

Symbol	**Purpose**
CJ_CCISIZE	Size of integer is 4 bytes (32 bits)
	Event group supports 32 event flags per group
CJ_ID	AMX id (handle) is a 32 bit unsigned integer
CJ_ERRST	AMX error codes are 32 bit signed integers
CJ_MINMSZ	Minimum AMX message size is 12 bytes
CJ_MAXMSZ	Default AMX message size is 12 bytes
CJ_MINKG	Minimum number of AMX message envelopes is 10
CJ_MINKS	Minimum Kernel Stack is 256 bytes
CJ_MINIS	Minimum Interrupt Stack is 256 bytes
CJ_MINTKS	Minimum task storage (including TCB) is 512 bytes
CJ_MINBFS	Minimum AMX buffer size is 8 bytes
CJ_MINUMEM	Minimum AMX memory block size is 16 bytes
CJ_MINSMEM	Minimum AMX memory section size is 128 bytes

Figure 1.4-1 AMX Design Constants

1.5 Launch Requirements

The ARM processor must be properly configured for use before AMX is launched. The manner in which this is accomplished will depend on your target hardware implementation and on the startup code provided with your C compiler.

AMX does not include bootstrap code to initialize the ARM processor. It is assumed that you will have a boot ROM present which configures the ARM for your specific hardware configuration and begins program execution at the entry to your C startup code.

During development, you may be using a ROM monitor provided by the processor vendor or by the toolset supplier. The ROM monitor automatically initializes the processor at power on. The monitor is then used to download your AMX application and start execution at the entry point to the C startup code. Eventually your *main* C program is called and AMX can be launched by your call to *cjkslaunch*.

Once your application has been tested, you may choose to replace the ROM monitor and the C startup code with your own initialization code. The manner in which you do this is outside the scope of this manual.

Operating Mode

The processor mode is determined by the mode bits in the Current Program Status Register (*CPSR*). AMX and all application procedures operate in supervisor mode. AMX never switches to user mode or system mode. When servicing interrupts, the AMX Interrupt Supervisor operates briefly in either IRQ or FIQ mode before switching to supervisor mode. When servicing fatal exceptions, the AMX Exception Supervisor will operate briefly in either abort mode or undefined mode before switching to supervisor mode.

Interrupt State

The ARM processor has two interrupt request input pins. Slow devices signal interrupt requests on the IRQ input. Fast devices signal interrupt requests on the FIQ input. AMX supports both types of interrupt requests.

Interrupts can be enabled or disabled on entry to AMX. Set the *I* and *F* bits in the *CPSR* to *1* to disable or to *0* to enable external interrupts. AMX will disable interrupts during its startup initialization. AMX will enable interrupts prior to calling your application Restart Procedures.

If you launch AMX with interrupts enabled, be sure that all interrupt sources are either disabled or masked off. You must not enable or unmask any interrupt source until you have installed an AMX Interrupt Service Procedure to properly service the device. This subject is described in more detail in Chapters 3 and 4.

ARM Stack Use

The ARM processor begins execution in supervisor mode with no active stack. Your bootstrap code or C startup code must establish a startup stack. Once AMX is launched, it abandons the startup stack.

AMX only uses the stacks allocated by you in your AMX System Configuration Module. The AMX Interrupt Stack is used to service the FIQ and IRQ interrupt exceptions. The AMX Kernel Stack is used for all other AMX services. Separate task stacks are used by your application tasks.

Instruction and Data Caching

The ARM architecture defines a System Control Coprocesser (CP15) which, if present, can provide for instruction and/or data cache implementation. AMX supports caches controlled by CP15 as defined by the ARM system architecture.

You must be aware that, on processors which utilize the ARM memory management unit (MMU) or memory protection unit (MPU), successful cache operation will depend on proper setup of the MMU or MPU. AMX does not manipulate the memory management subsystem. For example, if the MMU or MPU does not properly control cached access to memory and devices, you may find that device I/O reads and writes end up being cached, resulting in failure of the device to operate as expected.

Prior to launching AMX, you must ensure that the MMU or MPU is properly initialized to condition the instruction and data address translation logic to meet your hardware memory addressing specifications and caching requirements. Each KADAK board support module includes code to set up the MMU or MPU, if required, in order to run the AMX Sample Program on the board. You should review board support function *chbrdinit* to ensure that the initialization matches your specific board configuration.

The caches can be enabled or disabled prior to launching AMX. You can configure AMX to automatically enable the caches when AMX is launched. AMX will do so by calling the AMX cache support function *cjcfhwbcache*. Alternatively, you can configure AMX to ignore the caches during the launch.

For example, if you disable the caches in your main program and configure AMX to ignore the cache, you can simplify the initial testing of your application or overcome caching problems which may be encountered if your debugger cannot properly handle cached operation.

The AMX Sample Program is purposely configured such that AMX will not enable the caches during the launch, thereby avoiding possible cache related problems during your initial use of AMX in your hardware environment.

Note

AMX cache management services for the ARM processor are described in Appendix D.

Memory Management Unit (MMU)
Memory Protection Unit (MPU)

The ARM architecture defines a System Control Coprocesser (CP15) which, if present and fully implemented, includes a memory management unit (MMU) or memory protection unit (MPU). AMX does not support the memory management subsystem.

If you are using AMX on ARM processors which do not include the Coprocessor 15, this restriction does not apply. These processors do not implement the ARM memory management features and allow direct access to the full address space.

Your AMX application code and data must reside within the memory address ranges allowed by the particular ARM processor which you are using. The ARM MMU or MPU, if present, must be setup prior to or during the AMX launch. In most cases, your boot ROM or C startup code will configure the MMU or MPU for your specific hardware configuration prior to entry to your `main()` program.

Each KADAK board support module includes code to set up the MMU or MPU, if required, in order to run the AMX Sample Program on the board. You should review board support function `chbrdinit` to ensure that the initialization matches your specific board configuration.

Warning!

Do not enable the memory caches if the MMU or MPU has not been initialized to provide proper cached access to memory.

Big or Little Endian

AMX 4-ARM and AMX 4-Thumb are delivered ready for use with the little endian model in which the least significant byte of a word (long) is stored in the lowest byte address. Both kernels will also operate, without modification, on big endian hardware in which the most significant byte of a word (long) is stored in the lowest byte address. However, to use AMX on big endian hardware, you must first rebuild the AMX Library for big endian operation as described in Appendix D of the AMX User's Guide.

Execution State

AMX 4-ARM executes ARM instructions only. The *T* flag in the Current Program Status Register (*CPSR*) is always zero. AMX and all application code executes as ARM code.

AMX 4-Thumb executes both ARM and Thumb instructions according to the state of the *T* flag in the Current Program Status Register (*CPSR*). AMX and application code operate as ARM or Thumb code according to the rules defined in Chapter 2.1.

2. Program Coding Specifications

2.1 Programming Application Procedures

AMX 4-ARM executes unconditionally as ARM code. Any application procedure which AMX 4-ARM calls must also execute as ARM code. Any application which calls AMX must be executing as ARM code at the time of the call.

AMX 4-Thumb supports both ARM and Thumb instruction execution. The AMX task scheduler and Interrupt Supervisor operate as ARM code. All AMX procedures called from your application code operate as Thumb code.

Table 2.1-1 summarizes the required execution state for each type of application procedure which AMX 4-Thumb calls. Any application procedure which is called by AMX as Thumb code can switch to ARM code if required.

All procedures in the AMX 4-Thumb Library can be called from either Thumb or ARM code provided that you adhere to the Thumb to ARM interworking conventions dictated by your software development toolset. Be very careful when calling any AMX procedure which receives as a parameter a function pointer or a structure containing a function pointer. You must adhere to the restrictions imposed by your tools on the manner in which function pointers are passed and used.

State	**Application Procedure**
ARM	Interrupt Identification Procedure
ARM	Task Scheduling Hooks
ARM	Non-conforming Interrupt Service Procedure
Thumb	Task Procedure
Thumb	Task Termination Procedure
Thumb	Timer Procedure
Thumb	Time/Date Scheduling Procedure
Thumb	Restart Procedure
Thumb	Exit Procedure
Thumb	Interrupt Handler for a conforming ISP
Thumb	All customized error procedures in module `CJ422UF.C`

Figure 2.1-1 ARM v4T Application Execution State

Task Trap Handler

Unlike other processors, the ARM does NOT provide exceptions for faults such as arithmetic overflow, integer division by zero (no divide instruction) or array bounds violations. Consequently, AMX 4-ARM and AMX 4-Thumb do NOT support AMX task traps as found in other AMX implementations. If you are porting an AMX application from some other target processor, your Task Trap Handlers from that application will never be executed.

2.2 Task Scheduling Hooks

There are four critical points within the AMX Task Scheduler. These critical points occur when:

a task is started
a task ends
a task is suspended
a task is allowed to resume.

AMX allows a unique application procedure to be provided for each of these critical points. Pointers to your procedures are installed with a call to procedure *`cjkshook`*. You must provide a separate procedure for each of the four critical points. Since these procedures execute as part of the AMX Task Scheduler, their operation is critical. These procedures must be coded in assembler using techniques designed to ensure that they execute as fast as possible.

The AMX Task Scheduler calls each of your procedures with the same calling conventions.

Upon entry to your scheduling procedures, the following conditions exist:

The processor is executing in ARM state in supervisor mode.
Interrupts are disabled and must remain so.
Interrupts are disabled as dictated by your choice of interrupt model
(see Chapter 3.3).
The stack pointer in register *`sp`* references the task's stack.
The Task Control Block address is in register *`a2`*.
The return address is in register *`lr`*.
Registers *`a1`*, *`a2`*, *`a3`*, *`a4`*, *`v1`*, *`ip`* and *`lr`* are free for use.
The condition flags in the *`CPSR`* can be altered.
All other registers must be preserved.

Your procedures receive a pointer to the Task Control Block (TCB) of the task which is being started, ended, suspended or resumed. If you include AMX 4-ARM header file *`CJ402K.DEF`* (or AMX 4-Thumb header file *`CJ422K.DEF`*) in your assembly language module, you can reference the private region within the TCB reserved for your use as *`[a2, #XTCBUSER]`*.

Your procedures are free to temporarily use the task's stack.

3. The Processor Interrupt System

3.1 Operation

The ARM classifies all internal and external sources of interruption as exceptions. The processor automatically determines the cause of the exception and then branches directly to an appropriate exception specific procedure located at memory addresses called **Hard Vectors**. Each of the hard vectors is given an exception vector mnemonic which is defined in AMX 4-ARM header file *`CJ402KT.H`* (or AMX 4-Thumb header file *`CJ422T.H`*). Figure 3.1-1 summarizes these mnemonics.

The particular procedures which service internal or external device interrupt requests, software interrupts and processor faults are referred to as **exception service procedures**. Upon entry to each exception service procedure, the processor state is as defined by the ARM architecture.

AMX has default service procedures for every exception. Your Target Parameter File (see Chapter 4) uses the vector masks defined in Figure 3.1-1 to specify which of the possible exceptions you wish AMX to treat as fatal.

Usually one or the other or both of the FIQ and IRQ interrupts must be serviced by the AMX Interrupt Supervisor. All other exceptions, if serviced by AMX, will be assumed to be fatal.

Vector Name	**Vector Address**	**Vector Mask**	**Exception**
CJ_PRVNRES	*0x00*	*0x00000001*	Reset (Null pointer branches)
CJ_PRVNUI	*0x04*	*0x00000002*	Undefined Instruction
CJ_PRVNSWI	*0x08*	*0x00000004*	SWI (Software Interrupt)
CJ_PRVNIA	*0x0C*	*0x00000008*	Prefetch (Instruction) Abort
CJ_PRVNDA	*0x10*	*0x00000010*	Data Abort
CJ_PRVNADR	*0x14*	*0x00000020*	Address Exception
CJ_PRVNIRQ	*0x18*	*0x00000040*	IRQ (Normal External Interrupt)
CJ_PRVNFIQ	*0x1C*	*0x00000080*	FIQ (Fast External Interrupt)

Figure 3.1-1 AMX Hard Vector Definitions

Fatal Exceptions

Your Target Parameter File (see Chapter 4) identifies which of the exceptions the AMX Exception Supervisor is to service. The AMX Exception Supervisor always treats the exceptions which it services as fatal.

If AMX is allowed to service the reset exception, you will get a fatal exception whenever your application inadvertently attempts to jump to location *0* via a *NULL* pointer. Be forewarned that if your application uses a *NULL* pointer to store data at address *0*, the AMX reset exception handler will be destroyed and any subsequent branch to address *0* will yield unpredictable results.

If AMX cannot identify the device which generated an IRQ or FIQ interrupt, it generates a fatal exception using hard vector address *CJ_PRVNIRQ* (or *CJ_PRVNFIQ*) to identify the interrupt exception through which the spurious interrupt occurred.

If AMX has not been provided with an application handler to service a particular device interrupt, it generates a fatal exception using hard vector address *CJ_PRVNIRQ* (or *CJ_PRVNFIQ*) to identify the interrupt exception through which the device requested service.

If the IRQ or FIQ exception is not used in your application, your Target Parameter File can instruct AMX to treat spurious interrupts through the unused exception as fatal.

When a fatal exception is declared, AMX calls its Fatal Exception Procedure `cjksfatalexh` in AMX 4-ARM module `CJ402UF.C` (or AMX 4-Thumb module `CJ422UF.C`) identifying the exception and the machine state at the time of the exception. You are free to modify this procedure to meet the needs of your particular application.

The **Fatal Exception Procedure** is written in C as follows. Upon entry, the processor is in supervisor mode and the interrupt control bits in the Current Program Status Register (`CPSR`) are set to disable interrupts according to the AMX interrupt model which you are using.

```
#include "CJZZZ.H"                  /* AMX Headers                    */

void CJ_CCPP cjksfatalexh(
struct cjxregs *regp,               /* A(Register structure)          */
int vector)                         /* Hard vector address            */
                                    /*(see CJ_PRVNxxx definitions     */
                                    /* in Figure 3.1-1)               */
{
   :
   Process the error
   :
   }
```

The state of each supervisor mode register at the time of the fault is stored on the stack in an AMX register structure `cjxregs`. Parameter `regp` is a pointer to that structure. Structure `cjxregs` is defined in AMX 4-ARM header file `CJ402KT.H` (or AMX 4-Thumb header file `CJ422KT.H`). Note that the `pc` and `CPSR` register values in the register array reflect the state of the `pc` and `CPSR` registers immediately prior to the exception. Also note that if the exception occurred while in supervisor mode, the previous supervisor mode `lr` register content will have been lost and the value of the `lr` register stored in the array will also be the state of the `pc` register immediately prior to the exception.

If necessary, your Fatal Exception Procedure can access all of the banked registers for all processor modes by calling the AMX extended register access service procedure `cjcfmregst`.

If the Fatal Exception Procedure returns to AMX, AMX calls the Fatal Exit Procedure `cjksfatal` in AMX 4-ARM module `CJ402UF.C` (or AMX 4-Thumb module `CJ422UF.C`) with the AMX fatal exit code `CJ_FETRAP` signifying that a fatal exception has been detected and serviced but deemed unrecoverable.

Device Interrupts

The ARM processor provides two interrupt request exceptions, the fast FIQ interrupt and the normal IRQ interrupt. AMX supports both of these interrupts; your Target Parameter File (see Chapter 4) can configure AMX to service one or the other or both. Furthermore, AMX allows the use of external prioritizing hardware (such as an Intel 8259 Interrupt Controller) to permit concurrent or nested requests from multiple interrupting devices to be serviced in order of priority.

AMX will automatically create and install an interrupt dispatching exception service procedure into the IRQ and/or FIQ hard vectors for you, eliminating any need to program in ARM assembly language. All you have to do is identify for AMX, through your AMX Target Parameter File (see Chapter 4), how many devices are attached to each interrupt exception.

Interrupt Service

The external interrupt facility is fully enabled by setting both the *I* and *F* interrupt control bits in the Current Program Status Register (*CPSR*) to 0. The external interrupt facility is fully disabled by setting both the *I* and *F* interrupt control bits in the *CPSR* to 1. Note that when AMX disables (enables) interrupts, it always sets (clears) the *I* and *F* interrupt control bits according to the interrupt model being used by your application (see Chapter 3.3).

When an interrupt occurs, the processor switches to the IRQ or FIQ register bank. The return address + 4 (current Program Counter) is saved in the link register *R14_IRQ* (or *R14_FIQ*). The content of the Current Program Status Register (*CPSR*) prior to the interrupt is saved in register *SPSR_IRQ* (or *SPSR_FIQ*).

For IRQ interrupts, the *I* interrupt control bit is set to 1 thereby temporarily inhibiting another IRQ interrupt request. The *F* interrupt control bit is left unaltered allowing the possibility of immediate interrupts by higher priority FIQ devices.

For FIQ interrupts, the *I* and *F* interrupt control bits are both set to 1 thereby temporarily inhibiting all other IRQ and FIQ device interrupt requests.

The AMX exception service procedures for the IRQ and FIQ exceptions invoke the AMX Interrupt Supervisor to handle the interrupt. The Interrupt Supervisor calls an application Interrupt Identification Procedure (IIP) to determine which device generated the interrupt request. Using the interrupt number provided by the IIP, the AMX Interrupt Supervisor calls the application handler to service the particular device. To simplify this process, AMX provides an AMX Vector Table (see Chapter 3.2) which it uses to dispatch to individual device interrupt service routines.

Device Interrupt Service Procedures are of two types: conforming and nonconforming. A conforming Interrupt Service Procedure (see Chapter 3.4) adheres to the AMX Interrupt Supervisor rules and, by so doing, gains access to AMX task synchronization and communication services. Nonconforming Interrupt Service Procedures (see Chapter 3.5) bypass AMX completely.

3.2 AMX Vector Table and Interrupt Identification Procedure

AMX Vector Table

AMX maintains a Vector Table through which device interrupts are vectored for service. The AMX Vector Table is an array of pointers to service procedures for all of the internal or external devices which can generate interrupts. The entries in the AMX Vector Table are identified by vector numbers. The first entry in the Vector Table is assigned vector number 0.

AMX provides a set of *cjksixxxx* service procedures to allow you to dynamically access or modify entries in the AMX Vector Table. The AMX vector numbers must be used in all calls to these procedures to identify entries in the table.

The number of entries in the Vector Table and their assignment to devices is dictated by you, the application designer. The allocation is done with directives which the AMX Configuration Builder puts in your AMX Target Parameter File (see Chapter 4). A block of one or more vectors is allocated for each interrupt exception which you use.

For example, AMX vector number 1 might be allocated for the FIQ exception. Then a block of 16 vectors starting at AMX vector number 8 might be allocated for up to 16 devices attached through an interrupt controller to the IRQ interrupt exception. In this case, the AMX Vector Table would include 24 entries, seven of which are unallocated (vectors 0 and 2 to 7). Another example is given in Chapter 3.3.

Each entry in the Vector Table must be initialized with a valid pointer to a device service routine before the corresponding device is permitted to generate interrupts. AMX can be directed to automatically initialize any of the entries in the Vector Table during the AMX launch. You can also initialize entries with calls to the AMX *cjksixxxx* service procedures.

If AMX detects an interrupt for an uninitialized entry in its Vector Table, it generates a fatal exception using hard vector address *CJ_PRVNIRQ* (or *CJ_PRVNFIQ*) to identify the interrupt exception through which the interrupt for the unsupported device occurred.

Interrupt Identification Procedure (IIP)

When multiple devices generate interrupts through a single ARM exception vector, you must provide an **Interrupt Identification Procedure** which the AMX exception service procedure can call to determine the vector number assigned by you to the device which actually generated the interrupt exception. The IIP for each interrupt exception is defined by you in your AMX Target Parameter File (see Chapter 4). The IIP must be coded in ARM assembly language.

Upon entry to an **Interrupt Identification Procedure**, the following conditions exist:

> The processor is executing in ARM state in supervisor mode.
> For IRQ requests, IRQ interrupts are disabled and must remain so.
> For IRQ requests, the FIQ interrupt may be enabled or disabled.
> For FIQ requests, the IRQ and FIQ interrupts are disabled and must remain so.
>
> The stack pointer in register `sp` references the AMX Interrupt Stack.
> A register frame has been allocated on the stack and can be referenced via register `v1`.
> The return address is in register `lr`.
> Registers `a1`, `a2`, `a3`, `a4`, `ip` and `lr` have been saved and are free for use.
> The condition flags in the `CPSR` can be altered.
> All other registers must be preserved.

The Interrupt Identification Procedure must return the interrupt number for the particular device which generated the exception. The interrupt number is an index (`0` to `n-1`) into a block of `n` vectors in the AMX Vector Table allocated to the devices which are multiplexed through the ARM IRQ and/or FIQ interrupt exceptions.

The interrupt number is returned in register `a1` as follows:

`a1 = i`	Interrupt number (`0` to `n-1`)
`a1 = -1`	Ignore the interrupt
`a1 = -2`	Generate a fatal exception (unidentified interrupt request)

If you include AMX 4-ARM header file `CJ402K.DEF` (or AMX 4-Thumb header file `CJ422K.DEF`) in your assembly language module, you can use 5 words (20 bytes) in the register frame at `[v1,#XREG_R6]` for temporary storage.

Sample Interrupt Identification Procedures are included in the assembly language board support modules provided with AMX.

If your IIP returns to AMX with `a1 = -2`, AMX generates a fatal exception using hard vector address `CJ_PRVNIRQ` (or `CJ_PRVNFIQ`) to identify the interrupt exception through which the spurious interrupt occurred.

> Note
>
> When using the interrupt controller defined in the ARM Reference Peripherals Specification, the IIP can provide interrupt prioritization and nesting. Guidelines for doing so are provided in Appendix E.

3.3 AMX Interrupt Priority and Custom FIQ

The ARM family of processors provides limited interrupt priority ordering. The FIQ interrupt exception has priority over the IRQ interrupt exception. Service of an IRQ interrupt can be preempted by an FIQ request.

When multiple devices generate interrupts through a single ARM exception vector, external hardware (such as an Intel 8259 Interrupt Controller) can be used to arbitrate the requests and provide additional priority ordering. Some ARM implementations may include such an embedded interrupt controller to provide this feature.

When such a device is used for priority ordering, it is essential that the controller inhibit requests at priority `n` and lower until your application device handler has completed servicing the priority `n` device.

Note that there is no inherent relationship between the interrupt priority of a device and its vector number in the AMX Vector Table. However, for consistency with other AMX products, it is recommended that higher priority interrupts be assigned lower vector numbers. For example, the priority assigned to vector number 3 should be higher than that of vector number 4.

Interrupt Model

AMX can be configured to use one of the four possible interrupt models summarized in Figure 3.3-1. The full interrupt model allows the AMX Interrupt Supervisor to service all FIQ and IRQ interrupt requests as conforming AMX interrupts. The IRQ interrupt model restricts AMX interrupts to those generated by IRQ interrupt requests but still allows the FIQ interrupt to be used as a nonconforming interrupt or as a pseudo NMI. The FIQ interrupt model restricts AMX interrupts to those generated by FIQ interrupt requests and precludes use of the IRQ interrupt for any purpose.

The null interrupt model is of little use in an AMX application since neither the FIQ or IRQ interrupts are serviced by AMX. It is present for completeness.

The interrupt model dictates which bits (`I` and `F`) in the Current Program Status Register (`CPSR`) are used to disable (enable) interrupts.

Interrupt Model	**Interrupt Mask**	**Nonconforming IRQ**	**Nonconforming FIQ or pseudo NMI**
Full	`F` and `I`	no	no
IRQ	`I` only	no	allowed
FIQ	`F` only; `I=1`	cannot be used	no
Null	none	allowed	allowed

Figure 3.3-1 AMX Interrupt Models

Custom FIQ Interrupt Service

The FIQ interrupt on the ARM processor is designed specifically to provide fast, dedicated access to a custom device handler. This interrupt is especially suited for implementing block data transfers without the support of a hardware DMA interface.

To use the FIQ in this fashion, you simply instruct AMX in your Target Parameter File (see Chapter 4) to use the IRQ (or null) interrupt model in which AMX ignores the FIQ interrupt exception.

In this case, AMX will not alter the state of the FIQ interrupt control bit in the *CPSR*. When your application uses the AMX `cjcfdi` and `cjcfei` services to disable or enable interrupts, the FIQ interrupt state will not be affected. However, your application can still modify the FIQ interrupt state by altering the *CPSR* using the AMX `cjcfmodcpsr` service.

You must provide your own FIQ exception service procedure coded in ARM assembly language. Your FIQ exception service procedure will usually reside directly in the FIQ hard vector space as intended by ARM Ltd. However, if you wish, you can use the AMX service `cjcfhvwr` to install a branch to your procedure into the FIQ hard vector.

FIQ as a Pseudo Non-Maskable Interrupt

The ARM processor does NOT provide a non-maskable interrupt (NMI). However, the FIQ interrupt can be used as a pseudo NMI which CAN actually be inhibited by software.

To use the FIQ as a pseudo NMI, use the AMX Configuration Builder to put the pseudo NMI directive in your Target Parameter File (see Chapter 4). You must then use the IRQ (or null) interrupt model. At launch, AMX will enable the FIQ interrupt by clearing the *F* control bit in the Current Program Status Register (*CPSR*). Thereafter, AMX will not alter the state of the FIQ interrupt control bit in the *CPSR*.

When your application uses the AMX `cjcfdi` and `cjcfei` services to disable or enable interrupts, the FIQ interrupt state will not be affected. However, your application can still modify the FIQ interrupt state by altering the *CPSR* using the AMX `cjcfmodcpsr` service.

When the FIQ is used as a pseudo NMI, AMX does not install its FIQ interrupt exception service procedure into the FIQ hard vector. You must provide your own FIQ exception service procedure coded in ARM assembly language. You can use the AMX service `cjcfhvwr` to install a branch to your procedure into the hard vector.

Warning!

Because the FIQ interrupt can occur at any instant, including within critical sections of AMX, your custom FIQ handler or pseudo NMI handler cannot use any AMX service procedures for task communication.

3.4 Conforming ISPs

A conforming ISP consists of an ISP root and a device Interrupt Handler. The ISP root is created in your Target Configuration Module by the AMX Configuration Generator using the information provided in your Target Parameter File (see Chapter 4).

The address of the ISP root must be installed in the AMX Vector Table. You must provide a Restart Procedure or task which calls AMX procedure *`cjksivtwr`* or *`cjksivtx`* to install the ISP root pointer into the AMX Vector Table prior to enabling interrupt generation by the device.

The ISP root is the actual service procedure which is executed by the AMX Interrupt Supervisor once it has determined the source of the interrupt request.

The ISP root calls the device Interrupt Handler to dismiss the interrupt request and service the device. Upon return from the Interrupt Handler, the ISP root informs the Interrupt Supervisor that the interrupt service is complete. The Interrupt Supervisor either resumes execution at the point of interruption or invokes the Task Scheduler to suspend the interrupted task in preparation for a context switch. The path taken is determined by the actions initiated by your Interrupt Handler.

Interrupt Handlers can be written as C procedures with or without a single 32-bit formal parameter. The parameter, if needed, is identified in your definition of the ISP root in your Target Parameter File (see Chapter 4.3).

Upon entry to your Interrupt Handler written in C, the following conditions exist:

> For AMX 4-ARM, the processor is executing in ARM state.
> For AMX 4-Thumb, the processor is executing in Thumb state.
> The processor is executing in supervisor mode.
> For IRQ requests, IRQ interrupts are disabled.
> For IRQ requests, the FIQ interrupt may be enabled or disabled.
> For FIQ requests, FIQ and IRQ interrupts are disabled.
> The stack pointer in register *`sp`* references the AMX Interrupt Stack.

The Interrupt Handler can also be written in assembly language. Use assembly language if speed of execution is critical. Upon entry to an Interrupt Handler written in assembly language, the following additional conditions exist:

> Your Interrupt Handler parameter is in register *`a1`*.
> The return address is in register *`lr`*.
> A register frame has been allocated on the stack and can be referenced
> via register *`v1`*.
> Registers *`a1`*, *`a2`*, *`a3`*, *`a4`*, *`ip`* and *`lr`* are free for use.
> The condition flags in the *`CPSR`* can be altered.
> All other registers must be preserved.

If prioritized, nested interrupts are not supported, the Interrupt Handler must not enable interrupts. IRQ handlers must run with IRQ interrupts disabled. FIQ handlers must run with FIQ and IRQ interrupts disabled.

If prioritized, nested interrupts are supported by an interrupt controller, the Interrupt Handler can enable interrupts to allow nested interrupts. When using the full interrupt model, care must be taken to ensure that an FIQ handler never enables IRQ interrupts. Be sure to follow the guidelines presented in Appendix E.

The following examples illustrate how simple an Interrupt Handler can be.

```
/* The ISP root definition in the Target Parameter File is as follows:*/
/*          ...ISPC deviceisp,deviceih,17,0,0                         */
/*                                                                    */
/* The ISP root is given the public name deviceisp.                   */
/* The Interrupt Handler is named deviceih.                           */
/* The device has been assigned to AMX vector number 17.              */


void CJ_CCPP deviceih(void)
{
   local variables, if required
   :
   If (interrupt controller used AND nesting is supported AND
                                          nesting desired)
            Enable interrupts
   Clear the source of the interrupt request.
   Perform all device service.
   :
   }


/* Assume dcbinfo is some application device control block structure. */
/* Assume deviceXdcb is a structure variable defined as               */
/* "struct dcbinfo deviceXdcb;".                                      */
/*                                                                    */
/* The ISP root definition in the Target Parameter File is as follows:*/
/*          ...ISPC dcb_isp,dcb_ih,20,deviceXdcb,1                    */
/*                                                                    */
/* The ISP root is given the public name dcb_isp.                     */
/* The Interrupt Handler is named dcb_ih.                             */
/* The device has been assigned to AMX vector number 20.              */
/* deviceXdcb is the name of the public structure variable which      */
/* contains information about the specific device.                    */


void CJ_CCPP dcb_ih(struct dcbinfo *dcbp)
{
   local variables, if required
   :
   If (interrupt controller used AND nesting is supported AND
                                          nesting desired)
            Enable interrupts
   Use device control block pointer dcbp to access structure variable
   deviceXdcb to determine device addresses.
   Clear the source of the interrupt request.
   Perform all device service.
   :
   }
```

3.5 Nonconforming ISPs

Within the AMX programming environment, a nonconforming ISP is an ISP which bypasses AMX entirely in its service of the interrupting device. A nonconforming ISP cannot make use of any AMX services for task communication.

On the ARM processor, all interrupts serviced by the AMX Interrupt Supervisor are considered to be conforming. The only way to bypass AMX is to provide your own interrupt exception service procedure. For example, the custom FIQ exception service procedure and the pseudo NMI handler described in Chapter 3.3 are nonconforming ISPs.

A nonconforming ISP must NOT allow an interrupt from ANY higher priority conforming ISP. This restriction precludes your use of the IRQ interrupt as a nonconforming ISP when the FIQ interrupt model is used and the FIQ interrupt is used for conforming ISP service by AMX.

Figure 3.3-1 defines the AMX interrupt models and identifies the models which permit the FIQ or IRQ to be used as nonconforming interrupts.

3.6 Processor Vector Initialization

Whenever a processor fault, software interrupt or internal or external device interrupt occurs, the ARM processor unconditionally branches to a memory address called a hard vector. The code located at that address is called an exception service procedure. The hard vectors are located in the first 32 bytes of memory located at address 0. If necessary, you can use the AMX Configuration Builder to specify an alternate location for the hard vector table (see Chapter 4.2).

Your Target Parameter File defines whether the hard vectors are located in ROM or RAM. The Target Parameter File further qualifies whether or not AMX is allowed to modify the hard vectors if they are located in RAM.

If the hard vectors are declared to be alterable, AMX will install branches to the appropriate AMX exception service procedures into selected hard vectors. The specific hard vectors which will be affected are determined by the interrupt model you use and by your declaration of which hard vectors AMX must treat as fatal. These parameters are established in your Target Parameter File (see Chapter 4.2) using the AMX Configuration Builder.

AMX normally uses a short branch instruction in the hard vector table to branch to its exception handler. For this reason, the AMX Target Configuration Module which contains the AMX exception handlers must be located within +/- 32Mb of the hard vector table. If this constraint cannot be met, you can force AMX to use long branches through a hard address table, a linear array of eight pointers to the eight possible exception handlers. The address of the hard address table is provided in your Target Configuration Module (see Chapter 4.2). When used in this fashion, the AMX Target Configuration Module can be placed anywhere in memory and still be accessible by AMX.

Unalterable Exception Vectors

If the hard vector table is unalterable (in ROM, out of reach or simply constant by design), then it is your responsibility to initialize the hard vector content to meet your requirements. A branch, be it direct (short) or indirect (long), to a unique AMX exception service procedure must be installed into the hard vector for each exception for which AMX is to be responsible.

The AMX exception service procedures are located in an array beginning at entry point *cj_kdevt* in your Target Configuration Module. The AMX exception handler for hard vector *CJ_PRVNxxx* (see Figure 3.1-1) is located at address *cj_kdevt+CJ_PRVNxxx*. If AMX has been configured to ignore hard vector *CJ_PRVNxxx*, your hard vector table should not branch to *cj_kdevt+CJ_PRVNxxx*. If you do branch to such an unsupported handler, AMX will hang, looping forever on the instruction at that address.

An alternate approach is to provide a hard address table and install long branches in your hard vector table to dispatch indirectly through the table to the AMX exception handlers. Note that when a hard address table is provided, AMX initializes its content even if your hard vector table has been declared to be unalterable. Be aware that AMX only initializes the entries in the table for the exceptions for which it has been given responsibility.

4. Target Configuration Module

4.1 The Target Configuration Process

Every AMX application must include a **Target Configuration Module** which defines the manner in which AMX is to be used in your target hardware environment. The information in this file is derived from parameters which you must provide in your Target Parameter File.

The **Target Parameter File** is a text file which is structured according to the specification presented in Appendix A. You create and edit this file using the AMX Configuration Builder following the general procedure outlined in Chapter 16 of the AMX User's Guide. If you have not already done so, you should review that chapter before proceeding.

Using the Builder

When AMX is installed on your hard disk, the AMX Configuration Manager for Windows utility program and its related files are stored in directory *CFGBLDW* in your AMX installation directory. To start the Configuration Manager, double click on its filename, *CJ402CM.EXE* for AMX 4-ARM (or *CJ422CM.EXE* for AMX 4-Thumb). Alternatively, you can create a Windows shortcut to the manager's filename and then simply double click the shortcut's icon.

To create a new Target Parameter File, select New Target Parameter File from the File menu. The Configuration Manager will create a new, as yet unnamed, file using its default AMX target parameters. When you have finished defining or editing your target configuration, select Save As... from the File menu. The Configuration Manager will save your Target Parameter File in the location which you identify using the filename which you provide.

A good starting point is to copy one of the Sample Target Parameter Files *CJSAMTCF.UP* provided with AMX into file *HDWCFG.UP*. Choose the file for the evaluation board which most closely matches your hardware platform. Then edit the file to define the requirements of your target hardware.

To open an existing Target Parameter File such as *HDWCFG.UP*, select Open... from the File menu and enter the file's name and location or browse to find the file. When you have finished defining or editing your target configuration, select Save from the File menu. The Configuration Manager will rename your original Target Parameter File to be *HDWCFG.BAK* and create an updated version of the file called *HDWCFG.UP*.

To direct the Configuration Manager to use its Configuration Generator utility to produce an updated copy of your Target Configuration Module, say *HDWCFG.S*, select Generate... from the File menu. If necessary, the path to the template file required by the generator to create your Target Configuration Module can be defined using the Templates... command on the File menu.

The assembly language Target Configuration Module must be assembled as described in the toolset specific chapter of the AMX Tool Guide for inclusion in your AMX system. The assembler will generate error messages which exactly pin-point any inconsistencies in the parameters in your Target Parameter File.

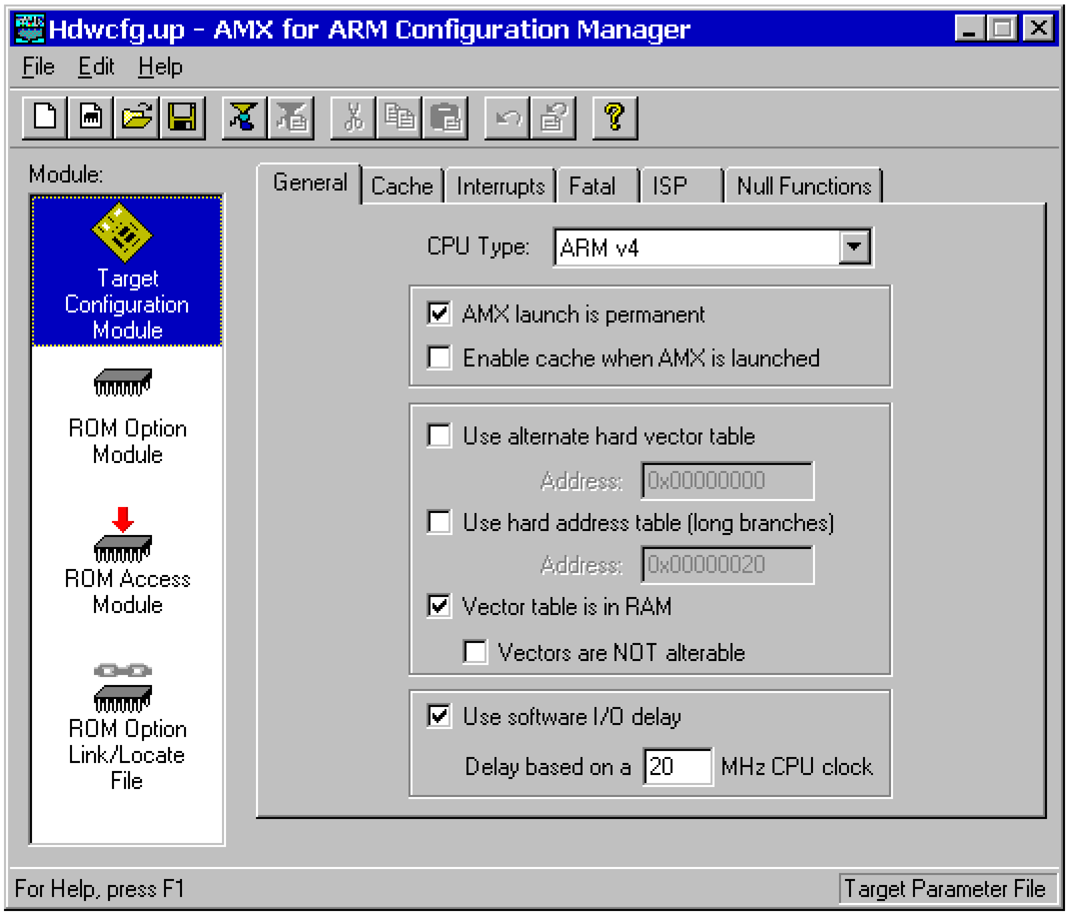
Screen Layout

Figure 4.1-1 illustrates the Configuration Manager's screen layout once you have begun to create or edit a Target Parameter File. The title bar identifies the Target Parameter File being created or edited. Below the title bar is the menu bar from which the operations you wish the Manager to perform can be selected. Below the menu bar is an optional Toolbar with buttons for many of the most frequently used menu commands.

At the bottom of the screen is the status bar. As you select menu items, a brief description of their purpose is displayed in the status bar. If the Configuration Manager encounters an error condition, it presents an error message on the status bar describing the problem and, in many cases, the recommended solution.

Along the left margin of the screen are a set of one or more selector icons. These icons identify the type of output files which the Manager's Configuration Generator will produce. The Target Configuration Module selector must be active to generate the Target Configuration Module.

The center of the screen is used as an interactive viewing window through which you can view and modify your target configuration parameters.


Hdwcfg.up - AMX for ARM Configuration Manager

CPU Type: ARM v4

[✓] AMX launch is permanent
[ ] Enable cache when AMX is launched

[ ] Use alternate hard vector table
Address: 0x00000000
[ ] Use hard address table (long branches)
Address: 0x00000020
[✓] Vector table is in RAM
[ ] Vectors are NOT alterable

[✓] Use software I/O delay
Delay based on a 20 MHz CPU clock

For Help, press F1

Figure 4.1-1 Configuration Manager Screen Layout

Menus

All commands to the Configuration Manager are available as items on the menus present on the menu bar. The **File menu** provides the conventional New, Open, Save and Save As... commands for creating and editing your Target Parameter File. It also provides the Exit command.

When the Target Configuration Module selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your Target Configuration Module. The path to the template file required by the generator to create this product can be defined using the Templates... command on the File menu.

The **Edit menu** provides the conventional Cut, Copy, Paste and Undo editing commands. It also includes an Undo Page command to restore the content of all fields on a property page to undo a series of unwanted edits to the page. The Toolbar is hidden or made visible using the View Toolbar command on the Edit menu.

The **Help menu** provides access to the complete AMX Configuration Manager reference manual. Context sensitive help is also available by pressing the F1 function key or clicking the **?** button on the Toolbar.

Field Editing

When the Target Configuration Module selector icon is the currently active selector, the Target Configuration Module's tabbed property sheet is displayed in the central region of the screen. Each tab provides access to a particular property page through which your target configuration parameters can be declared. For instance, if you select the ISP tab, the Configuration Manager will present an ISP definition window (property page) containing all of the parameters you must provide to completely define an Interrupt Service Procedure.

Some fields are boolean options in which all you can do is turn the option on or off by checking or unchecking the associated check box.

Some fields are option fields in which you must select one of a set of options identified with radio buttons. Click on the option button which meets your preference.

Other fields may require numeric or text entry. Parameters are entered or edited in these fields by typing new values or text to replace the current field content. Only displayable characters can be entered. New characters which you enter are inserted at the current cursor position in the field. Right and left arrow, backspace and delete keys may be used to edit the field.

When you are altering a numeric or text field, you tell the Configuration Manager that you are finished editing the field by striking the Enter key. At that point, the Configuration Manager checks the numeric value or text string that you have entered for correctness in the context of the current field. If the value or text string that you have entered is invalid, an error indication is provided on the status bar at the bottom of the screen suggesting how the fault should be corrected.

The Tab and Shift-Tab keys can also be used to complete the editing of a field and move to the next or previous field.

If you have modified some of the fields on a property page and then decide that these modified values are not correct, use the Undo Page command on the Edit menu or Toolbar to force the Configuration Manager to restore the content of all fields on the page to the values which were in effect when you moved to that property page.

When you go to save your Target Parameter File or prepare to move to another property page, the Configuration Manager will validate all parameters on the page which you are leaving. If any parameters are incomplete or inconsistent with each other, you will be forced to fix the problem before being allowed to proceed.

Add, Edit and Delete Objects

Separate property pages are provided to allow your definition of one or more objects such as ISPs or null functions. Pages of this type include a list box at the left side of the property page in which the currently defined objects are listed. At the bottom of the list box there may be a counter showing the number of objects in the list and the allowable maximum number of such objects.

Also below the list are two control buttons labeled Add and Delete. If the allowable maximum number of objects is 0 or if all such objects have already been defined, the Add button will be disabled. If there are no objects defined, the Delete button and all other fields on the page will be disabled.

To add a new object, click on the Add button. A new object with a default identifier will appear at the bottom of the list and will be opened ready for editing. When you enter a valid identifier for the object, your identifier will replace the default in the object list.

To edit an existing object's definition, double click on the object's identifier in the object list. The current values of all of that object's parameters will appear in the property page and the object will be opened ready for editing.

To delete an existing object, click on the object's identifier in the object list. Then click on the Delete button. Be careful because you cannot undo an object deletion.

The objects in the object list can be rearranged by dragging an object's identifier to the desired position in the list. You cannot drag an object directly to the end of the list. To do so, first drag the object to precede the last object on the list. Then drag the last object on the list to precede its predecessor on the list.

4.2 Target Configuration Parameters

General Parameters

The General Parameter window allows you to define the general operating characteristics of your AMX system within your target hardware environment. The layout of the window is shown in Figure 4.1-1 in Chapter 4.1.

CPU Type

Identify your processor architecture by selecting a processor from the available list. This parameter is used to condition AMX to accommodate the operating characteristics of a particular processor or architecture. If you are using AMX 4-Thumb, only those processors which support the ARM v4T or v5T architecture will be included in the list. The supported list of processors includes but is not limited to:

ARM v4 (or *v5*)	ARM v4 or v5 architecture
ARM v4 Thumb	ARM v4T architecture (v4 with Thumb enhancements)
ARM v5 Thumb	ARM v5T architecture (v5 with Thumb enhancements)
ARM7TDMI	NEC ARM7TDMI processor (test chip)
ARM710	VLSI ARM710 processor
ARM720T (et al)	ARM720T, ARM740T
ARM910T (et al)	ARM910T, ARM920T, ARM922T, ARM940T, ARM926EJ, ARM946E, ARM966E
ARM1020E (et al)	ARM1020E, ARM1022E, ARM1026EJ
ARM1136J (et al)	ARM1136J, ARM1156T2, ARM1176JZ
Atmel AT91	Atmel AT91 processor family
Cirrus Logic EP73xx	Cirrus Logic EP73xx processor family
Samsung S3C4510	Samsung S3C4510 processor
StrongARM SA-110	Intel (Digital Semiconductor) StrongARM® SA-110
StrongARM SA-1100	Intel (Digital Semiconductor) StrongARM® SA-1100
XScale 80200	Intel 80200 XScale™ processor
XScale 80321	Intel 80321 XScale™ processor
Freescale i.MX1	Freescale i.MX1 processor
Freescale i.MX21	Freescale i.MX21 processor

If you are using a custom ASIC with an embedded ARM core, select the processor by choosing the ARM architecture which that core implements.

Note that AMX 4-Thumb only supports processors that are compliant with the ARM v4T or v5T architecture specification.

AMX Launch

Most AMX applications are such that once AMX is launched the application runs forever. For such applications, check this box. If your AMX launch is to be temporary, uncheck this box. In this case, you will be able to shut down your AMX application and return to your main program from which AMX was launched.

Enable Cache at Launch

If the processor or architecture indicated by field `CPU Type` has cache control, then, before launching AMX, you must initialize the Memory Management Unit (MMU) or Memory Protection Unit (MPU) in Coprocessor 15 to condition the memory subsystem to meet the caching requirements of your system.

When AMX is launched, if this box is checked, AMX will enable the processor instruction and data caches by calling the AMX cache support function *`cjcfhwbcache`*.

When AMX is launched, if this box is unchecked, AMX will not alter the state of the processor instruction or data caches.

If the processor or architecture indicated by field `CPU Type` has no cache control, leave this box unchecked.

Alternate Hard Vector Table

AMX assumes that the ARM hard vector table is located at memory address 0. If your memory configuration locates the hard vector table at some other address or somehow provides a shadow vector table at some RAM address, you can check this box to force AMX to use that alternate table. The associated parameter is the hexadecimal memory address of the alternate table.

Hard Address Table

Exception service procedures must reside within +/-32Mb of the ARM hard vector table. If your memory configuration does not permit AMX code to be located within that region, AMX will be unable to service exceptions.

To overcome this restriction, you can check this box and provide the location of a hard address table which AMX can use to force a long branch to each service procedure. The hard address table is a block of 32 bytes of long-aligned, alterable memory which must be located within +/-4096 bytes of the base of the ARM hard vector table. The associated parameter is the hexadecimal memory address of the hard address table.

Vectors in RAM

In most cases, the processor Exception Vector Table will be located in alterable RAM at address 0 or at some alternate address provided by you. Therefore check this box.

If your processor Exception Vector Table is in ROM, leave this box unchecked. In this case, you must initialize the ROM hard vectors for AMX use as directed in Chapter 3.6.

Vectors Not Alterable

Even if the processor Exception Vector Table will be located in RAM, you can still prevent AMX from altering it. To do so, check this box. In this case, be sure to initialize the hard vectors for AMX use as directed in Chapter 3.6.

Software I/O Delay

AMX provides a device I/O delay procedure *cjcfhwdelay* which is used by AMX board support modules and sample device drivers to provide the necessary delay between sequential references to a device I/O port. Such delay is often required to accommodate long device access times when operating at very high processor clock frequencies.

Check this box to adjust the AMX software delay loop to match your hardware requirements. Enter your best estimate of the processor's effective instruction execution frequency. AMX will use this parameter to derive the loop count needed to provide a one microsecond delay.

For example, if your processor executes at 40 MHz with no wait states for instruction fetches and one clock cycle per instruction, enter a CPU clock frequency of 40 MHz.

If you are able to detect the processor frequency at run time, then you can dynamically adjust this I/O delay procedure to match your target hardware without reconfiguring your AMX application. To do so, enter a CPU frequency of 0 MHz. In this case, your *main()* program must install the processor frequency value into *long* variable *cjcfhwdelayf* prior to launching AMX.

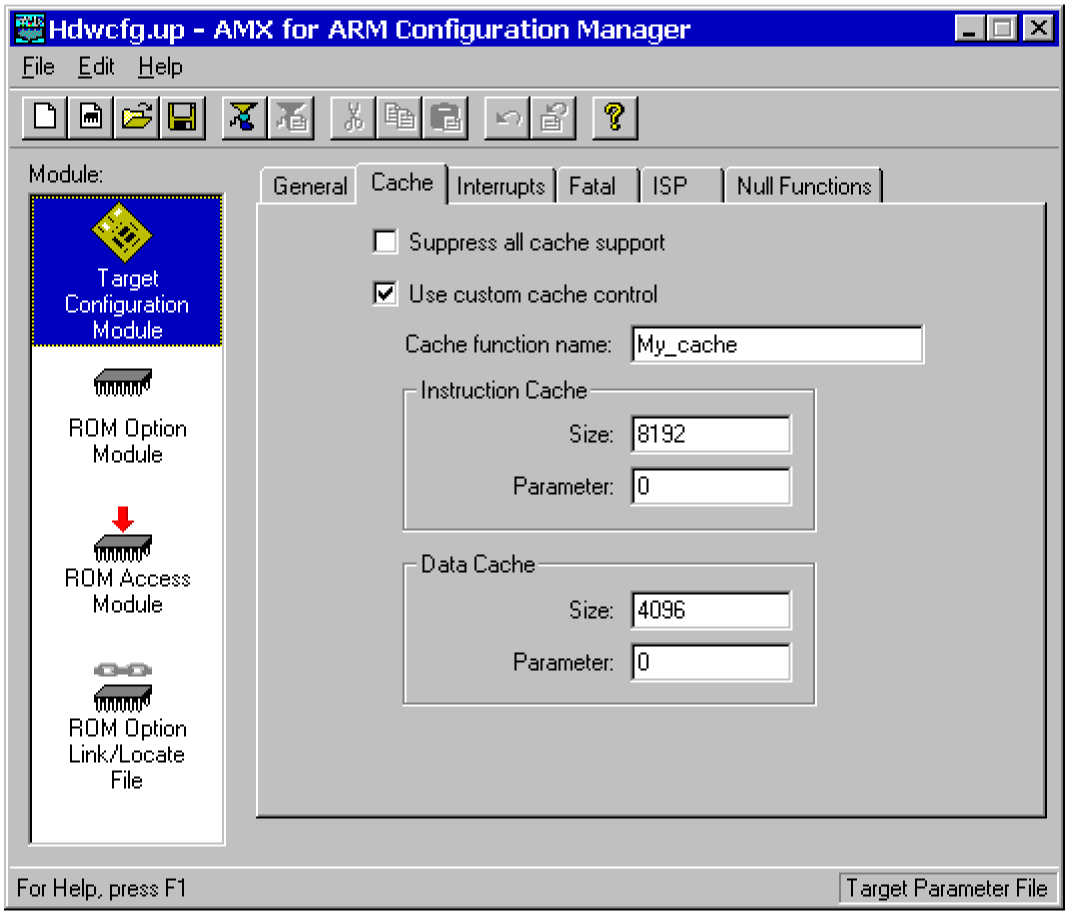
Leave this box unchecked if you want the I/O delay procedure *cjcfhwdelay* to produce no delay beyond that inherent in the procedure call and return.

Cache Parameters

The Target Configuration Module includes cache support functions *cjcfhwicache*, *cjcfhwbcache* and *cjcfhwdcache* tailored for the specific processor or architecture identified by the CPU Type on the General Parameters page. These functions call one of the AMX cache control procedures *chXXXcache* in the AMX library to enable or disable the instruction and/or data cache.

The Cache Parameter window allows you to suppress cache support, provide your own cache service procedures or customize those provided by AMX for your target hardware. The layout of the window is shown below.

The most common use of the cache customization feature is to inject alternate cache sizes into one of the standard AMX cache control functions in order to support a new processor variant.



[ ] Suppress all cache support

[✓] Use custom cache control

Cache function name: My_cache

Instruction Cache
Size: 8192
Parameter: 0

Data Cache
Size: 4096
Parameter: 0

Suppress Cache Support

Check this box to completely suppress AMX cache support. The AMX cache support functions *cjcfhwXcache* will exist but will do nothing. If you suppress cache support, AMX will not be able to enable the cache at launch time if you have so requested on the General Parameters page. Furthermore, the custom cache control features will also be disabled.

Custom Cache Control

Check this box if you wish to customize the AMX cache control functions or force the cache support functions *cjcfhwXcache* to call an alternate cache control function. This process is described in Appendix D.3.

Cache Control Function Name

Enter the name of an AMX cache control function which you wish to adapt for your own use. Alternatively, enter the name of your own custom cache control function. The function must be prototyped as follows:

```
   void CJ_CCPP YOURcache(unsigned int command,
                          unsigned long icsize,
                          unsigned long icparam,
                          unsigned long dcsize,
                          unsigned long dcparam);
```

The *`command`* parameter defines the cache operation. It is a bit mask identifying the cache or caches to be affected and the operation to be performed. The bit masks are defined as follows:

`0x80000000L`	Select the instruction cache
`0x40000000L`	Select the data cache
`0x00000001L`	*`0/1`* = disable/enable the selected caches

The remaining four parameters which your cache control function receives are those provided by you in the Instruction Cache and Data Cache screen fields:

`icsize`	Instruction cache size
`icparam`	Instruction cache parameter
`dcsize`	Data cache size
`dcparam`	Data cache parameter

AMX Cache Control Parameters

The AMX cache control functions use parameter *`icsize`* to define the total size, in bytes, of the instruction cache. Parameter *`icparam`* is used to identify the instruction cache block (cache line) characteristics.

The AMX cache control functions use parameter *`dcsize`* to define the total size, in bytes, of the data cache. Parameter *`dcparam`* is used to identify the data cache block (cache line) characteristics.

The specific values for these parameters will vary depending upon the cache type and how it must be controlled. The default values required for each type of AMX cache control function are provided in Appendix D.2. In many cases, you will be able to adapt one of the AMX cache control functions to meet your cache requirements by simply adjusting these parameter values.

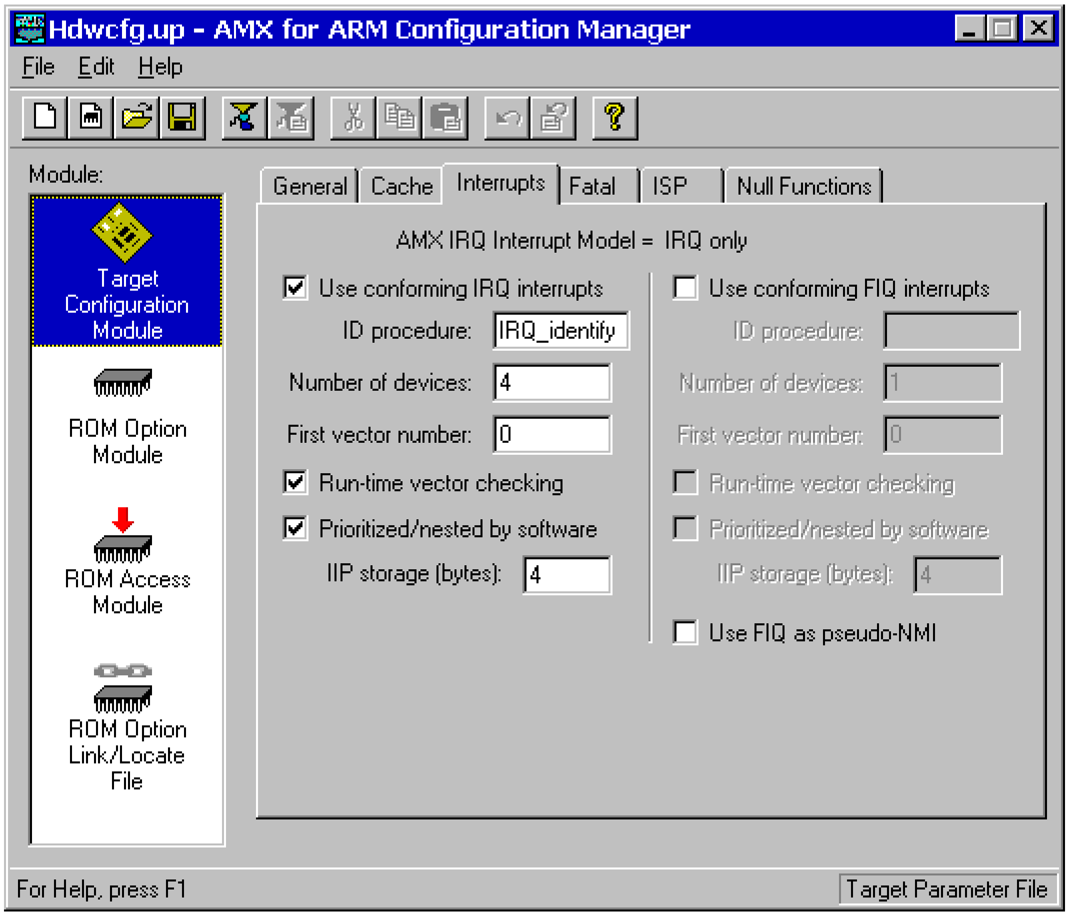
Your Cache Control Parameters

If you provide your own custom cache control function, the interpretation of the four cache control parameters is left entirely to your function.

Interrupt Model

The Target Configuration Module defines the AMX interrupt model you intend to use. The model is determined by checking the IRQ and/or FIQ box in the Interrupt Model window. The layout of the window is shown below.

The interrupt model window is divided into two similar panes, one for the IRQ interrupts and one for the FIQ interrupts. If you check the IRQ or FIQ box, you must fill in the remaining parameters in the corresponding pane.



AMX IRQ Interrupt Model = IRQ only

[✓] Use conforming IRQ interrupts
ID procedure: IRQ_identify
Number of devices: 4
First vector number: 0
[✓] Run-time vector checking
[✓] Prioritized/nested by software
IIP storage (bytes): 4

[ ] Use conforming FIQ interrupts
ID procedure:
Number of devices: 1
First vector number: 0
[ ] Run-time vector checking
[ ] Prioritized/nested by software
IIP storage (bytes): 4
[ ] Use FIQ as pseudo-NMI

Interrupt Model Selection

The **full interrupt model** is selected by checking both the IRQ and FIQ boxes. The FIQ and IRQ interrupt exceptions are both serviced by the AMX Interrupt Supervisor. All interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) both the *F* and *I* bits in the *CPSR*.

The **IRQ interrupt model** is selected by checking the IRQ box and leaving the FIQ box unchecked. The IRQ interrupt exception is serviced by the AMX Interrupt Supervisor. All IRQ interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) the *I* bit in the *CPSR*. The *F* bit is not altered.

The **FIQ interrupt model** is selected by checking the FIQ box and leaving the IRQ box unchecked. The FIQ interrupt exception is serviced by the AMX Interrupt Supervisor. All FIQ interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) the *F* bit in the *CPSR*. The *I* bit is set to unconditionally inhibit IRQ interrupts. The IRQ interrupt exception cannot be used for any purpose.

The **null interrupt model** is selected by leaving both the IRQ and FIQ boxes unchecked. AMX Interrupt Supervisor will not service the IRQ or FIQ interrupt exceptions. This model is rarely used since no AMX clock or other AMX interrupt devices are supported.

Dedicated Device Interrupt

If a **single device** generates an interrupt through the IRQ (FIQ) interrupt exception vector, fill in the IRQ (FIQ) exception definition as follows.

Since there is only one device, leave the ID procedure field blank (empty) and leave the remaining check boxes unchecked. You will not be able to leave the Interrupt Model window if these unused fields are improperly filled. An error message describing the fault will appear in the status bar.

Number of Devices

Set the number of devices to one (1).

First Vector Number

Enter the vector number in the AMX Vector Table which you wish to reserve for the device.

When the full interrupt model is used, the AMX vectors assigned for the IRQ and FIQ devices must not overlap.

Multiplexed Device Interrupts

If **multiple devices** generate interrupts through the IRQ (FIQ) interrupt exception vector, fill in the IRQ (FIQ) exception definition as follows.

ID Procedure

You must provide the name of an Interrupt Identification Procedure which AMX can call to determine the device requiring service. Your procedure returns a number from *0* to *n-1* identifying which of the *n* possible devices generated the interrupt currently under service.

It is allowable to have fewer than *n* physical devices capable of generating interrupts. For example, your Interrupt Identification Procedure might support as many as 8 devices even though only 3 devices are actually able to generate the IRQ (FIQ) interrupt exception.

Number of Devices

Set the number of devices to *n* where *n* is the total number of devices which your Interrupt Identification Procedure is capable of identifying.

First Vector Number

A block of *n* vectors in the AMX Vector Table must be reserved for the devices which generate interrupts through the IRQ (FIQ) interrupt exception. Enter the base vector number of the block of *n* AMX vectors which you wish to assign to these devices.

The interrupt identification number (*0* to *n-1*) provided by your Interrupt Identification Procedure will be added to this base value to derive the AMX vector number for the device requesting service.

When the full interrupt model is used, the two sets of AMX vectors assigned for the IRQ and FIQ devices must not overlap.

Run-time Vector Checking

The AMX Interrupt Supervisor can check that the derived vector number lies within the allowable range in the AMX Vector Table. If it does not, AMX calls the Fatal Exception Handler indicating that an unidentified interrupt exception was detected via hard vector *CJ_PRVNFIQ* or *CJ_PRVNIRQ*.

To enable vector number checking by the AMX Interrupt Supervisor, check this box. To disable vector number checking and thereby reduce interrupt service overhead, leave the box unchecked.

Prioritized / Nested by Software

If you are using an ARM processor which implements the interrupt controller defined in the ARM Reference Peripherals Specification, you can enhance your IRQ or FIQ Interrupt Identification Procedure (IIP) to prioritize the interrupts and to allow support for nested interrupts. Similarly, if you are using your own interrupt prioritization circuitry, you can enhance your IRQ or FIQ Interrupt Identification Procedure (IIP) to support nested interrupts. Instructions for doing so are provided in Appendix E.

Most IIPs of this kind will require some storage to retain the interrupt mask history in order to successfully unravel the nested interrupts. For these IIPs, AMX will allocate an extra block of storage on its Interrupt Stack for each IRQ (FIQ) interrupt. Your IIP can use the storage region to save and restore interrupt mask information on entry and exit from each interrupt.

If your IIP supports prioritization and nesting, check this box and enter the amount of history storage that your IIP requires.

Most of the Interrupt Identification Procedures in the board support modules provided with AMX support interrupt nesting. If you use one of these IIPs, you must check this box and allocate 4 bytes of history storage for its use.

If you leave this box unchecked, nesting of IRQ (FIQ) interrupts will not be supported. AMX will service each interrupt to completion before servicing another interrupt from the same exception.

Use FIQ as Pseudo Non-Maskable Interrupt (NMI)

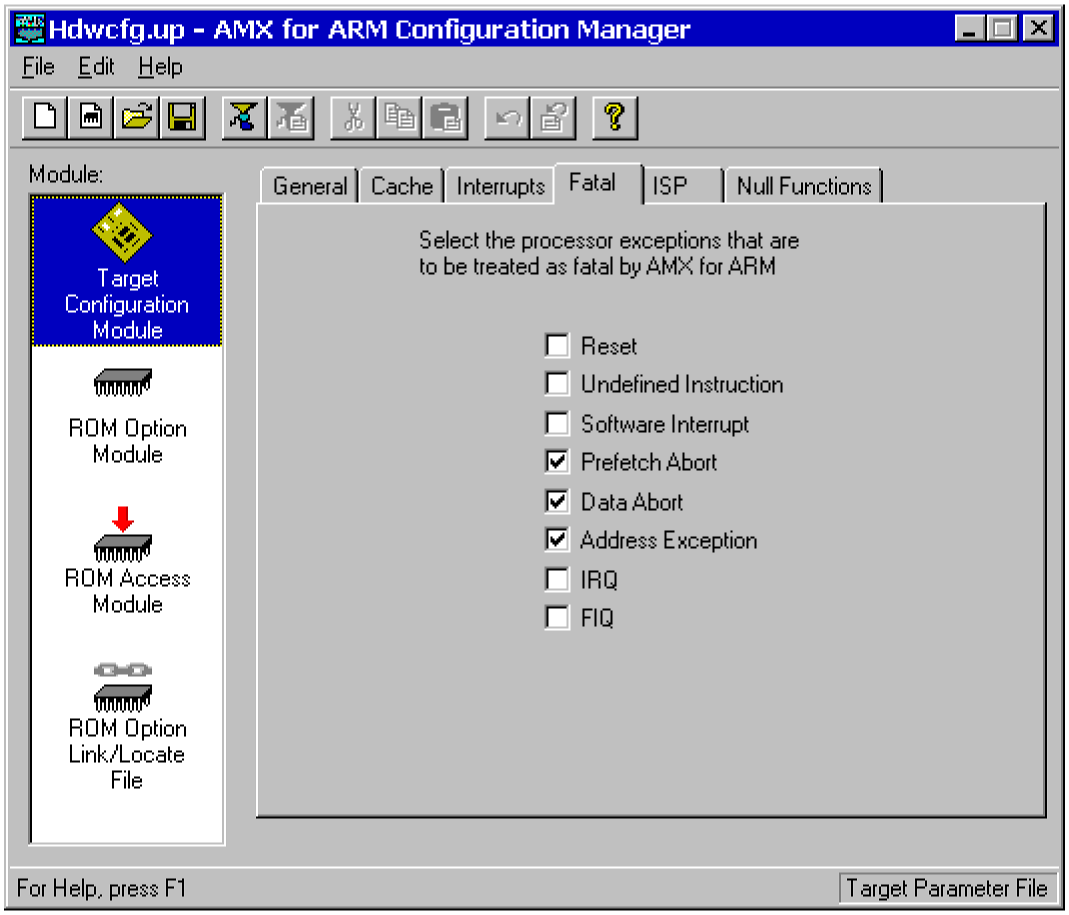
If your AMX application uses the IRQ or null interrupt model, the FIQ interrupt exception will not be used by the AMX Interrupt Supervisor. In this case, the FIQ interrupt exception can be used as a pseudo non-maskable interrupt as described in Chapter 3.3.

Check this box if you wish to use the FIQ interrupt exception as a pseudo non-maskable interrupt. Otherwise, leave the box unchecked.

Fatal Exceptions

The Target Configuration Module defines the processor exceptions which are to be serviced by AMX and treated as fatal. These exceptions are specified by you by checking the appropriate boxes in the Fatal Exception window. The layout of the window is shown below.

This example allows AMX to service the prefetch abort, data abort and address exceptions as fatal exceptions. This example leaves the reset, undefined instruction and SWI exceptions free for use by a debugger and leaves the IRQ and FIQ exceptions free for use by the AMX Interrupt Supervisor or your application.



Select the processor exceptions that are to be treated as fatal by AMX for ARM

- [ ] Reset
- [ ] Undefined Instruction
- [ ] Software Interrupt
- [✓] Prefetch Abort
- [✓] Data Abort
- [✓] Address Exception
- [ ] IRQ
- [ ] FIQ

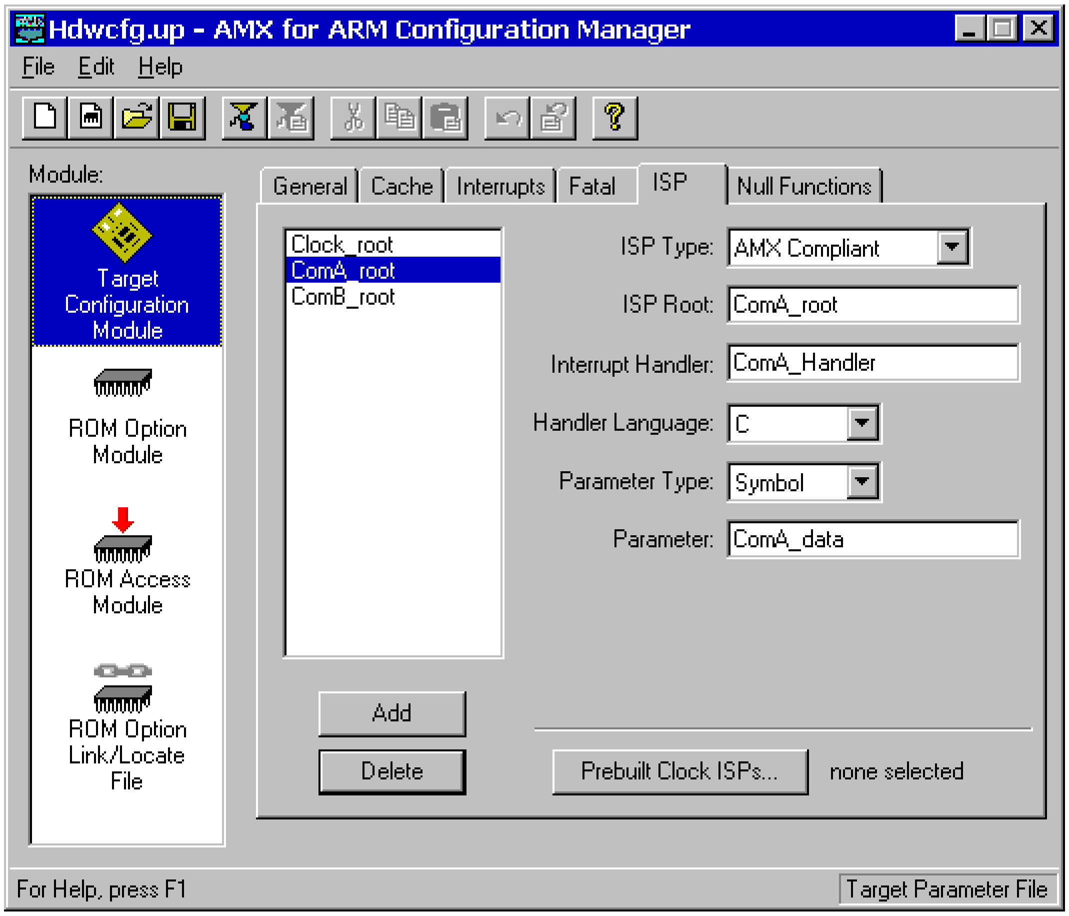
4.3 Interrupt Service Procedure (ISP) Definitions

Your Target Configuration Module must include a device ISP root for each **conforming ISP** which you intend to use in your application. The ISP roots are constructed for you by the AMX Configuration Builder from ISP descriptions which you enter in the ISP Definition window. The layout of the window is shown below.

To add an ISP definition, click on the Add button. A new ISP with a default ISP root name of `---New---` will appear at the bottom of the ISP list and will be opened ready for editing. When you enter a name for the ISP root, it will replace the default name in the ISP list.

To edit an existing ISP definition, click on the name of the ISP root in the ISP list. The ISP definition will appear in the property page and will be opened ready for editing.

To delete an existing ISP definition, click on the name of the ISP root in the ISP list. Then click on the Delete button. Be careful because you cannot undo an ISP deletion.



Clock_root
ComA_root
ComB_root

ISP Type: AMX Compliant
ISP Root: ComA_root
Interrupt Handler: ComA_Handler
Handler Language: C
Parameter Type: Symbol
Parameter: ComA_data

none selected

ISP Type

At least one of your ISPs must service a clock interrupt which provides AMX with its fundamental clock tick at the frequency and resolution defined in your AMX System Configuration Module. To define your custom **clock ISP**, choose Clock Handler from the pull down list. An alternate fast clock ISP can be provided by choosing Fast Clock Handler as described in Chapter 4.4. Other AMX clock drivers can be selected from the list presented when you click the Prebuilt Clock ISPs... button.

All other application ISPs must be conforming AMX ISPs which you define by choosing AMX Compliant from the pull down list.

ISP Root

Edit the default name `---New---` to provide the name you wish to give to the ISP root. The ISP root name is used to identify ISPs in the ISP list.

The ISP root is a function created by the AMX Configuration Builder in your Target Configuration Module. The function entry point is declared with a public symbol defined with the name you provide. The name must be unique and must conform to the symbol naming conventions of your assembler.

Interrupt Handler

Enter the name of your device Interrupt Handler which will clear the device interrupt request and service the device. This is the name of the procedure which will be called from the ISP root by the AMX Interrupt Supervisor once the interrupt source has been identified and the machine state preserved according to the conditions which existed at the time of the interrupt. Your Interrupt Handler must be coded as described in Chapter 3.4.

If your Interrupt Handler is coded in C, you may have to add a leading or trailing underscore to the Interrupt Handler name which you enter in order to meet the C function naming conventions of your C compiler.

Handler Language

Your Interrupt Handler can be coded in C or assembly language. Identify the language in which your Interrupt Handler is written by picking C or Assembly from the pull down list.

Interrupt Handler Parameter

Your Interrupt Handler can be coded to receive a 32-bit parameter every time it is called. The Parameter Type field is a pull down list used to identify what kind of parameter, if any, your Interrupt Handler expects. If your Interrupt Handler has no need for a parameter, set the Parameter Type to **(none)**.

If your Interrupt Handler expects a numeric parameter, set the Parameter Type to **Value** and enter the required unsigned, 32-bit hexadecimal numeric value into the Parameter field.

If your Interrupt Handler parameter must be a pointer to a variable or function, set the Parameter Type to **Symbol** and enter the name of the variable or function into the Parameter field. The parameter must be a text string giving the name of a public symbol (variable or function) defined in some module in your AMX application. The symbol's 32-bit value, as resolved by your linker, will be passed to your Interrupt Handler as its parameter.

Prebuilt Clock ISPs

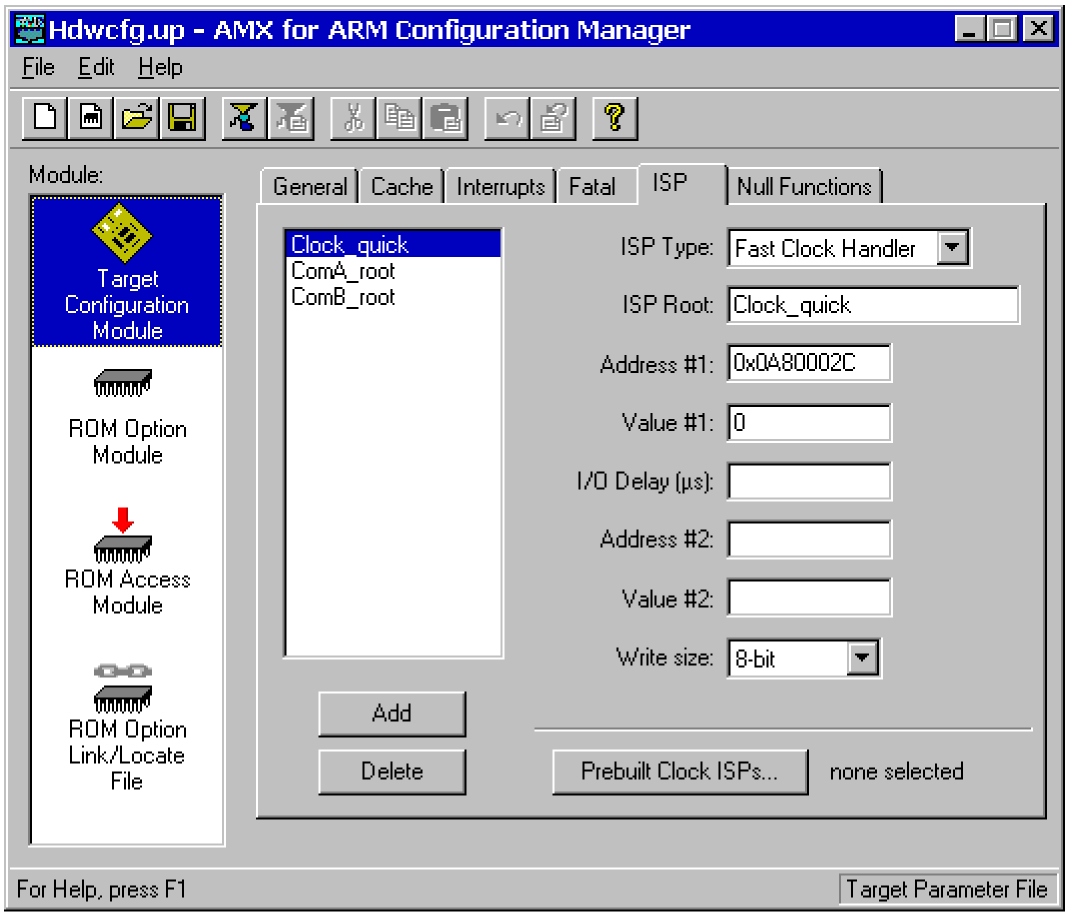
Clock drivers are provided with AMX for several common programmable interval timers. In some cases, the AMX clock ISP can be prebuilt in your Target Configuration Module. To select one of these clocks, click on the Prebuilt Clock ISPs... button. In the dialog box which is presented, check the box for the particular clock driver which you wish to use. If you do not wish to use a prebuilt clock ISP, leave all boxes unchecked.

4.4 Defining a Fast Clock ISP

At least one of your ISPs must service a clock interrupt which provides AMX with its fundamental clock tick at the frequency and resolution defined in your AMX System Configuration Module. For many applications, your clock ISP will just be a standard AMX conforming ISP defined in the ISP Definition window. It is distinguished from all other ISPs by picking Clock Handler as its ISP Type.

Rarely does the Interrupt Handler for your AMX clock ISP have to do anything except dismiss the clock interrupt request. This is frequently accomplished by simply writing a command to a device I/O port. For such clocks, the AMX Configuration Builder lets you create a fast clock ISP without having to write any code at all.

To create a fast clock ISP, go to the ISP Definition window, click on the Add button and select Fast Clock Handler as the ISP Type. Then fill in the description of the operating characteristics of your clock device. The layout of the window is shown below.



Clock_quick
ComA_root
ComB_root

ISP Type: Fast Clock Handler
ISP Root: Clock_quick
Address #1: 0x0A80002C
Value #1: 0
I/O Delay (µs):
Address #2:
Value #2:
Write size: 8-bit

none selected

ISP Type

Your fast clock ISP is identified as such by selecting Fast Clock Handler from the pull down list.

ISP Root

Edit the default name `---New---` to provide the name you wish to give to your fast clock ISP root. The ISP root name is used to identify your fast clock ISP in the ISP list.

The ISP root is a function created by the AMX Configuration Builder in your Target Configuration Module. The function entry point is declared with a public symbol defined with the name you provide. The name must be unique and must conform to the symbol naming conventions of your assembler.

Clock Service

Your clock device will be serviced as follows:

> Write Value #1 to the device port at memory Address #1.
> Delay for the number of µs defined as I/O Delay (µs).
> Write Value #2 to the device port at memory Address #2.

Address and Value

Each address parameter specifies the 32-bit, hexadecimal value of an absolute memory address which, when referenced as an *n*-bit value, is decoded by your target hardware as a reference to your clock device. Each value parameter is an *n*-bit, hexadecimal value which must be written to the device port specified by the associated address in order to dismiss the clock interrupt.

If your clock device only requires that one value be written to one device port, leave fields Address #2 and Value #2 blank (empty).

I/O Delay (µs)

Your target hardware may not operate correctly if two sequential device I/O references are issued at the processor's instruction execution speeds. If this is the case, you can force the fast clock ISP to inject a delay of *n* µs between the I/O device references by entering a non-zero value into this field.

If your clock device requires no delay or only requires that one value be written to one device port, leave the I/O Delay field blank (empty).

Write Size

From the pull down list, select the number of bits (8, 16 or 32) which must be written to the clock device. The least significant *n* bits of each value will be written to the device.

4.5 Null Functions

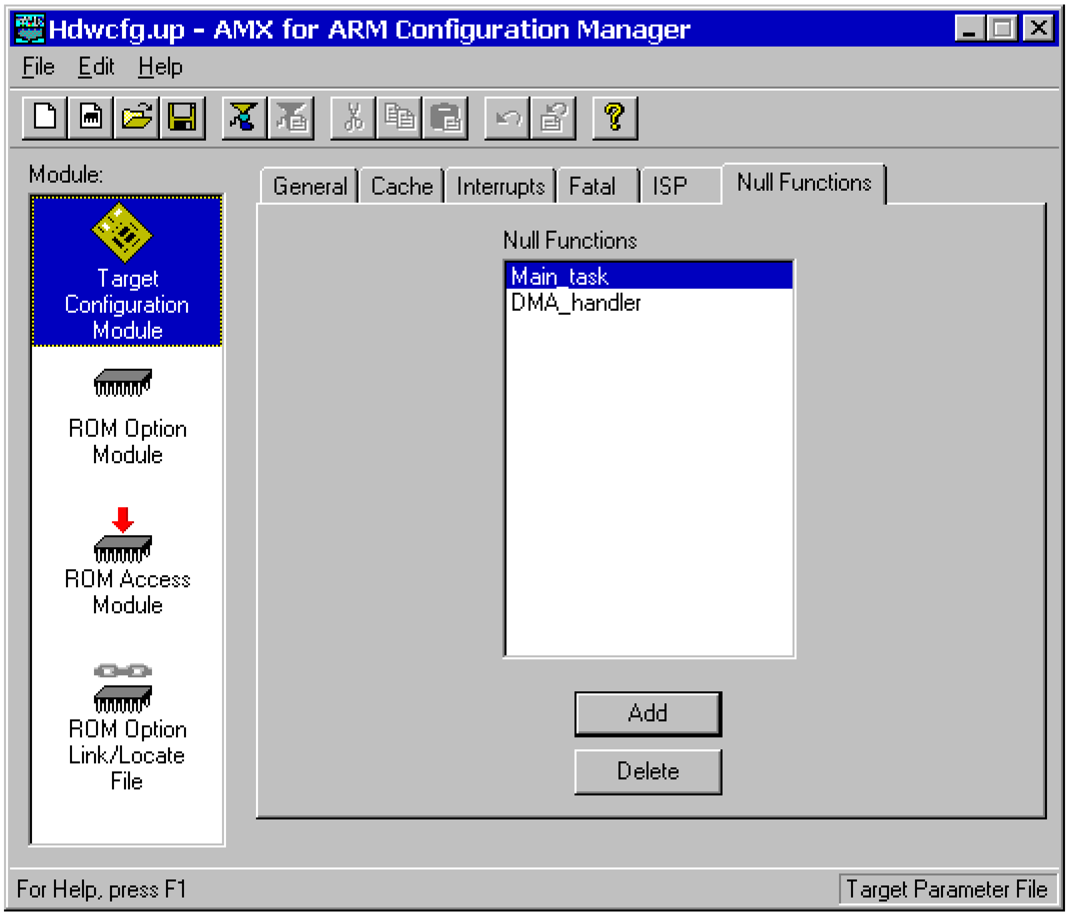
Occasionally, while developing an AMX application, it can be very convenient to be able to create software functions to satisfy your program link requirements without having to create the final version of these functions. For example, if your AMX System Configuration Module references a Restart Procedure and a task procedure which do not yet exist, you will have to create them in order to successfully link your system.

Such functions are called null functions because they do nothing. Such functions can be specified by you in the Null Function window whose layout is shown below.

To add a null function, click on the Add button. A new function named `---New---` will appear at the bottom of the list of functions. Click on the name in the list and edit it to meet your needs.

To edit the name of a null function, double click on its name in the list and edit it to meet your needs.

To delete a null function, click on its name in the list and then click on the Delete button.



Null Functions

Main_task
DMA_handler

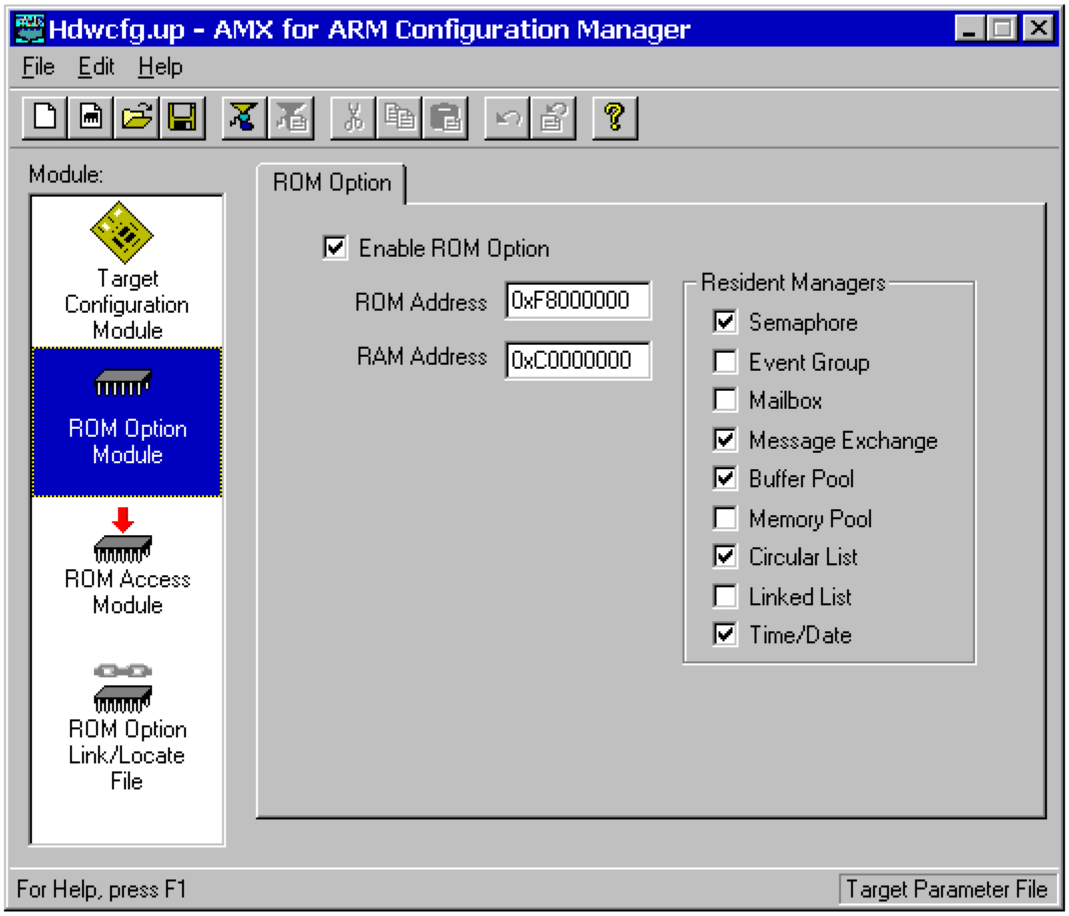
4.6 ROM Option Parameters

The AMX ROM Option allows the subset of AMX and its managers required by your application to be linked together without any application code to form a separate AMX ROM image. The resulting AMX ROM can be located anywhere in your memory configuration. Your AMX application is then linked with a ROM Access Module which provides access to AMX and its managers in the AMX ROM.

The AMX ROM Option Module defines the subset of AMX and its managers which you wish to commit to the AMX ROM. This module is assembled and linked with the AMX Library to create that ROM. The AMX ROM Option Link/Locate Specification File is used to link and locate the ROM image as described in the toolset dependent chapter of the AMX 4-ARM or 4-Thumb Tool Guide.

The AMX ROM Access Module provides your AMX application with access to the AMX ROM. This module is assembled and linked with your AMX application.

To access the ROM Option window, use the AMX Configuration Builder to open your Target Parameter File. From the selector list, pick the ROM Option Module selector making it the active selector. The layout of the window is shown below.



[✓] Enable ROM Option

ROM Address: 0xF8000000

RAM Address: 0xC0000000

Resident Managers
- [✓] Semaphore
- [ ] Event Group
- [ ] Mailbox
- [✓] Message Exchange
- [✓] Buffer Pool
- [ ] Memory Pool
- [✓] Circular List
- [ ] Linked List
- [✓] Time/Date

Enable ROM Option

By default, the ROM Option feature is disabled. Check this box to enable the feature. You can disable the feature by removing the check from the box.

ROM Address

You must define the absolute physical ROM address at which the AMX ROM image is to be located. This address is dictated by you according to your hardware requirements. Enter the address value as an unsigned 32-bit hexadecimal number. The ROM memory address must be long aligned.

RAM Address

You must define the absolute physical RAM address of a block of 32 bytes reserved for use by AMX. This address is dictated by you according to your hardware requirements. Enter the address value as an unsigned 32-bit hexadecimal number. The RAM memory address must be long aligned.

Resident Managers

Check the boxes which identify the AMX managers which you wish to commit to the AMX ROM. If you do not want a particular manager to be in the ROM, leave the corresponding box unchecked.

> Warning!
>
> If your AMX ROM was created without a particular manager, then an AMX fatal exit will occur if your system attempts to access that manager.

5. Clock Drivers

5.1 Clock Driver Operation

You must provide a clock driver as part of your AMX application so that AMX can provide timing services. AMX clock drivers are provided with AMX for the timer chips used on the boards with which AMX has been tested. These drivers are ready for use and can be installed as described in Chapter 5.3.

An AMX clock driver consists of three parts: an initialization procedure, a clock Interrupt Service Procedure (ISP) and an optional shutdown procedure.

Clock Startup

The **clock initialization procedure** must configure the real-time clock to operate at the frequency defined in your AMX System Configuration Module. It can then install the pointer to the clock ISP root into the AMX Vector Table and start the clock.

Care must be taken to ensure that clock interrupts do not occur until the clock is properly configured and the pointer to the clock ISP root is present in the AMX Vector Table.

Your AMX application will not have any AMX timing services until your clock initialization procedure, say `clockinit`, has been executed. The first opportunity for `clockinit` to execute occurs when AMX begins to execute your Restart Procedures. It is recommended that your `clockinit` procedure be inserted into your list of Restart Procedures at the point at which you wish the clock to be enabled during the launch.

Although it is not recommended, there is nothing to prohibit you from deferring the starting of your clock by having some application task call your `clockinit` procedure.

The clock drivers provided with AMX illustrate how to install and start several different real-time clocks. You should be able to pattern your clock initialization procedure after the chip support procedure `chclockinit` in one of the AMX clock driver source files `CHxxxxT.C`.

Clock Interrupts

A real-time clock used with the ARM processor will interrupt on either the IRQ or FIQ pin. If the clock is one of several devices interrupting on the pin, your Interrupt Identification Procedure must sense that your clock is the device demanding service and provide the AMX Interrupt Supervisor with the interrupt identification number assigned by you to the clock. Once the AMX Interrupt Supervisor has determined that your clock is the interrupting device, it dispatches through its Vector Table to your clock ISP.

The **clock ISP** consists of an ISP root and an Interrupt Handler. The ISP root is called by the AMX Interrupt Supervisor in response to the clock interrupt request. The ISP root calls the clock Interrupt Handler to dismiss the clock interrupt request. Your clock ISP must be defined as a conforming ISP of type Clock Handler as described in Chapter 4.3.

In some cases you may be able to create a fast clock ISP which has an ISP root but no Interrupt Handler. In this case, it is the ISP root which dismisses the clock interrupt request. Such a clock ISP is defined to be a conforming ISP of type Fast Clock Handler as described in Chapter 4.4.

In other cases you may be able to pick one of the prebuilt AMX clock ISPs which has an ISP root but no Interrupt Handler. In this case, it is the ISP root which dismisses the clock interrupt request. Such a clock ISP is selected from the list which is accessed via the Prebuilt Clock ISPs... button.

It is the ISP root which informs AMX that a hardware clock tick has occurred. When you define your clock ISP, your definition of the ISP as a Clock Handler (or Fast Clock Handler) or your selection of a prebuilt clock ISP ensures that the ISP is recognized by AMX as the source of its fundamental clock tick operating at the frequency and resolution defined in your AMX System Configuration Module.

Clock Shutdown

The **clock shutdown procedure** stops the clock in preparation for an AMX shutdown following a temporary launch of AMX. If AMX is launched for permanent execution, there is no need for a clock shutdown procedure.

If you intend to launch AMX for temporary execution, insert your clock shutdown procedure, say *`clockexit`*, into your list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown. Usually that will require that *`clockexit`* be the last Exit Procedure in the list because, once you stop your clock, AMX timing services will no longer be available.

The clock drivers provided with AMX illustrate how to disable several different real-time clocks. You should be able to pattern your clock shutdown procedure after the chip support procedure *`chclockexit`* in one of the AMX clock driver source files *`CHxxxxT.C`*.

5.2 Custom Clock Driver

The easiest way to create a custom clock driver is by example. Assume that the counter/timer which you intend to use for your AMX clock is characterized as follows:

The I/O base address of the clock is at *0xF0000100*.
The clock interrupt is generated through the IRQ external interrupt exception.
Your Interrupt Identification Procedure assigns interrupt identification number 2 to the clock, thereby also assigning it to vector number 2 in the AMX Vector Table.
The clock interrupt is dismissed by writing bit pattern *0x08* to the clock register at its base address plus 4.

The Interrupt Handler for an assembly language conforming clock ISP for such a device could be coded as follows:

```
         EXPORT   clockih
clockih  EQU      .
;
; receives a1 = ISP root parameter = A(clock base)
; return address is in lr
;
         MOV      a2,#8                  ; a2 = bit pattern = 0x08
         STRB     a2,[a1, #4]            ; Dismiss interrupt
         MOV      pc,lr                  ; Return from ARM code
         or
         BX       lr                     ; Return from Thumb code
```

Create a clock ISP root for the clock as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*clockroot*
Interrupt Handler:	*clockih*
Handler Language:	Assembly
Parameter Type:	Value
Parameter:	*0xF0000100*

Note that you could just as easily create a fast clock ISP root for this simple clock as described in Chapter 4.4 avoiding the need to create the Interrupt Handler *clockih*. Use the following parameters in your definition of the fast clock ISP.

ISP Type:	Fast Clock Handler
ISP Root:	*clockroot*
Address #1:	*0xF0000104*
Value #1:	*0x08*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	8-bit

The clock initialization procedure for this custom clock driver could be coded in C as follows. Insert procedure *clockinit* into your list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch.

```
void CJ_CCPP clockroot(void);             /* External clock ISP root  */

void CJ_CCPP clockinit(void)
{
      /* Inhibit clock interrupts                                     */
      /* Configure clock for correct frequency                        */

      /* Install pointer to clock ISP root into AMX Vector Table      */
      cjksivtwr(2, (CJ_ISPPROC)clockroot);

      /* Start clock and enable clock interrupts                      */
      }
```

5.3 AMX Clock Drivers

AMX clock drivers are provided with AMX for the timer chips used on the boards with which AMX has been tested. These drivers are ready for use as described in this chapter. The clock drivers are delivered in **chip support** source files having names of the form *CHnnnnT.C* where *nnnn* identifies the particular clock chip. The clock chip support procedures are named *chxxxxxxx*.

5.3.1 ARM Reference Clock Driver

The ARM Reference Peripherals Specification (document ARM DDI 0062D) defines a standard timer peripheral which can be incorporated into any ARM processor. The AMX clock driver for this device is ready for use on the ARM Development Board equipped with an ARM7TDMI test chip operating at 20 MHz. It is configured to use timer 1 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH0062T.C*.

Board support module *ADB7T.S*, or your equivalent, must be used to configure the interrupt controller to allow the timer to generate an IRQ interrupt which is assigned interrupt identification number 4 which maps directly to AMX vector number 4. To use this module on the ARM Development Board equipped with an ARM7TDMI test chip, you must select the IRQ interrupt model as described in Chapter 4.2 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*chadb7tirq*
Number of devices:	*16*
First vector number:	*0*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *chadb7tirq* in board support module *ADB7T.S* assigns interrupt identification numbers 4 and 5 to reference timers 1 and 2 respectively. Since the reference timers interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 4 and 5 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *chadb7tirq* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *CH0062T.C*, assemble board support source module *ADB7T.S* and link the resulting object modules with the rest of your AMX application.

To use the AMX clock driver for the ARM reference clock, you must create a fast clock ISP root as described in Chapter 4.4. Use the following parameters in your definition of the clock ISP. Note that the interrupt from timer 1 is cleared by writing one byte of any value to the timer clear register at device address *0x0A80000C*. The interrupt from timer 2 is cleared by writing one byte to device address *0x0A80002C*.

ISP Type:	Fast Clock Handler
ISP Root:	*ch0062clk*
Address #1:	*0x0A80000C* for timer 1; *0x0A80002C* for timer 2
Value #1:	*0*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	8-bit

The board support module *ADB7T.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH0062T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the Reference Clock Driver

If you wish to use ARM reference timer 2, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH0062T.C* and recompile the module. Edit instructions are included in the file.

If you wish to use ARM reference timer 2, assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the reference clock driver to a different hardware platform, you will also have to edit and assemble the board support module *ADB7T.S*. Edit instructions are included in the file.

The board support module *ADB7T.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your board.

Other Boards

Variants of clock driver module *CH0062T.C* are provided for use on other boards with which AMX has been tested. The fast clock parameters are specified in the AMX Sample Program Target Parameter File *CJSAMTCF.UP* for each board. Separate board support modules include processor specific support for the following boards:

Use module *ADB9T.S* for the ARM940T on the ARM Development Board.
Use module *IAP966E.S* for the ARM966E (et al) on the ARM Integrator/AP Board.
Use module *ICP920T.S* for the ARM920T (et al) on the ARM Integrator/CP Board.

5.3.2 21285 Core Logic Clock Driver

The AMX clock driver for the 21285 Core Logic controller is ready for use on the Intel (Digital Semiconductor) EBSA-285 Evaluation Board. It is configured to use timer 1 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CH21285T.C`*.

The 21285 timer generates an IRQ interrupt which is assigned interrupt identification number 4. Board support module *`EBSA285.S`*, or your equivalent, must be used to service the 21285 Core Logic interrupt controller. To use this clock driver on the EBSA-285 Evaluation Board, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*`chsa110irq`*
Number of devices:	*`16`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *`chsa110irq`* in board support module *`EBSA285.S`* assigns interrupt identification numbers 4 through 7 to 21285 timers 1 through 4 respectively. Since the 21285 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 4 through 7 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *`chsa110irq`* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *`CH21285T.C`*, assemble board support source module *`EBSA285.S`* and link the resulting object modules with the rest of your AMX application.

Note

Since the StrongARM processor does not support Thumb execution, the 21285 Clock Driver for the EBSA-285 board is NOT provided with AMX 4-Thumb.

To use the AMX 21285 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the 21285 clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *EBSA285.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH21285T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the 21285 Clock Driver

If you wish to use a different 21285 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH21285T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the 21285 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *EBSA285.S*.

The board support module *EBSA285.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.3 SA-1100 Clock Driver

The AMX clock driver for the SA-1100 processor's internal timers is ready for use on the Intel (Digital Semiconductor) Brutus SA-1100 Evaluation Platform. It is configured to use timer 0 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CH1100T.C`*.

The SA-1100 timer generates an IRQ interrupt which is assigned interrupt identification number 26. Board support module *`SA1100EP.S`*, or your equivalent, must be used to service the SA-1100 interrupt controller. To use this clock driver on the SA-1100 Evaluation Platform, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*`chsa1100irq`*
Number of devices:	*`32`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked
Use FIQ as pseudo-NMI	Checked

Note that the standard Interrupt Identification Procedure *`chsa1100irq`* in board support module *`SA1100EP.S`* assigns interrupt identification numbers 26 through 29 to SA-1100 timers 0 through 3 respectively. Since the SA-1100 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 26 through 29 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *`chsa1100irq`* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *`CH1100T.C`*, assemble board support source module *`SA1100EP.S`* and link the resulting object modules with the rest of your AMX application.

Note

Since the StrongARM processor does not support Thumb execution, the SA-1100 Clock Driver for the Brutus SA-1100 board is NOT provided with AMX 4-Thumb.

To use the AMX SA-1100 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the SA-1100 clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *SA1100EP.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH1100T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the SA-1100 Clock Driver

If you wish to use a different SA-1100 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH1100T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the SA-1100 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *SA1100EP.S*.

The board support module *SA1100EP.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.4 SA-1110 Clock Driver

The AMX clock driver for the SA-1110 processor's internal timers is ready for use on the Intel (Digital Semiconductor) SA-1110 Development Platform. It is configured to use timer 0 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH1110T.C*.

The SA-1110 timer generates an IRQ interrupt which is assigned interrupt identification number 26. Board support module *SA1111BS.S*, or your equivalent, must be used to service the SA-1110 interrupt controller. To use this clock driver on the SA-1110 Development Platform, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*chsa1110irq*
Number of devices:	*128*
First vector number:	*0*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked
Use FIQ as pseudo-NMI	Checked

Note that the standard Interrupt Identification Procedure *chsa1110irq* in board support module *SA1111BS.S* assigns interrupt identification numbers 26 through 29 to SA-1110 timers 0 through 3 respectively. Since the SA-1110 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 26 through 29 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *chsa1110irq* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *CH1110T.C*, assemble board support source module *SA1111BS.S* and link the resulting object modules with the rest of your AMX application.

Note

Since the StrongARM processor does not support Thumb execution, the SA-1110 Clock Driver for the SA-1110 board is NOT provided with AMX 4-Thumb.

To use the AMX SA-1110 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the SA-1100 clock ISP on the list provided via the Prebuilt Clock ISPs... button. Although the SA-1100 clock driver is selected, the SA-1110 clock driver will be used as long as you have identified the SA-1110 as your CPU Type on the Configuration Manager's General property page.

The board support module *SA1111BS.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH1110T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the SA-1110 Clock Driver

If you wish to use a different SA-1110 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH1110T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the SA-1110 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *SA1111BS.S*.

The board support module *SA1111BS.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.5 Atmel AT91 Clock Driver

The AMX clock driver for the Atmel AT91 Timer/Counter is ready for use on the Atmel AT91EB40 Evaluation Board. It is configured to use timer 0 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CHAT91T.C*.

The AT91 timer 0 generates an IRQ interrupt which is assigned interrupt identification number 4. Board support module *EB40.S*, or your equivalent, must be used to service the AT91 interrupt controller. To use this clock driver on the AT91EB40 Evaluation Board, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*chat91irq*
Number of devices:	*32*
First vector number:	*0*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *chat91irq* in board support module *EB40.S* assigns interrupt identification numbers 4, 5 and 6 to AT91 timers 0, 1 and 2 respectively. Since the AT91 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 4, 5 and 6 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *chat91irq* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *CHAT91T.C*, assemble board support source module *EB40.S* and link the resulting object modules with the rest of your AMX application.

To use the AMX AT91 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the AT91 clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *EB40.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CHAT91T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the AT91 Clock Driver

If you wish to use a different AT91 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CHAT91T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number or port the AT91 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *EB40.S*.

The board support module *EB40.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

Other Boards

Variants of clock driver module *CHAT91T.C* are provided for use on other boards with which AMX has been tested. Separate board support modules include specific support for the following boards:

Use module *EB42.S* for the AT91 processor on the AT91EB42 Evaluation Board.
Use module *SAM7S.S* for the AT91 processor on the AT91SAM7S-EK Evaluation Kit.

5.3.6 Samsung S3C4510 Clock Driver

The AMX clock driver for the Samsung S3C4510 32-Bit Timer is ready for use on the ARM Ltd. Evaluator-7T board. It is configured to use timer 0 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CH4510T.C`*.

The S3C4510 timer 0 generates an IRQ interrupt which is assigned interrupt identification number 10. Board support module *`EVAL7T.S`*, or your equivalent, must be used to service the S3C4510 interrupt controller. To use this clock driver on the ARM Evaluator-7T board, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*`ch4510irq`*
Number of devices:	*`32`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *`ch4510irq`* in board support module *`EVAL7T.S`* assigns interrupt identification numbers 10 and 11 to S3C4510 timers 0 and 1 respectively. Since the S3C4510 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 10 and 11 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *`ch4510irq`* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *`CH4510T.C`*, assemble board support source module *`EVAL7T.S`* and link the resulting object modules with the rest of your AMX application.

To use the AMX S3C4510 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the S3C4510 clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *EVAL7T.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH4510T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the S3C4510 Clock Driver

If you wish to use a different S3C4510 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH4510T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the S3C4510 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *EVAL7T.S*.

The board support module *EVAL7T.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.7 Intel IOP321 Clock Driver

The AMX clock driver for the 32-bit timer in the Intel XScale™ 80321 I/O Processor (IOP321) is ready for use on the Intel IQ80321 Evaluation Platform. It is configured to use timer 0 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CH321T.C`*.

The IOP321 timer 0 generates an IRQ interrupt which is assigned interrupt identification number 9. Board support module *`EVP321.S`*, or your equivalent, must be used to service the IOP321 interrupt controller. To use this clock driver on the Intel IQ80321 Evaluation Platform, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*`chevp321irq`*
Number of devices:	*`32`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *`chevp321irq`* in board support module *`EVP321.S`* assigns interrupt identification numbers 9 and 10 to IOP321 timers 0 and 1 respectively. Since the IOP321 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 9 and 10 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *`chevp321irq`* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *`CH321T.C`*, assemble board support source module *`EVP321.S`* and link the resulting object modules with the rest of your AMX application.

To use the AMX IOP321 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the 80321 clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *EVP321.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH321T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the IOP321 Clock Driver

If you wish to use a different IOP321 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH321T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or AMX vector number, move the clock to the FIQ interrupt or port the IOP321 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *EVP321.S*.

The board support module *EVP321.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.8 Cirrus Logic EP7312 Clock Driver

The AMX clock driver for the Cirrus Logic EP7312 General Purpose Timer is ready for use on the Cogent Computer Systems, Inc. CSB238 Single Board Computer. It is configured to use timer 1 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CH7312T.C`*.

The EP7312 timer 1 generates an IRQ interrupt which is assigned interrupt identification number 4. Board support module *`CSB7312.S`*, or your equivalent, must be used to service the EP7312 interrupt controller. To use this clock driver on the Cogent CSB238 board, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming FIQ interrupts	Checked
ID procedure:	*`ch7312fiq`*
Number of devices:	*`8`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Unchecked
IIP storage (bytes):	(unused)

Use conforming IRQ interrupts	Checked
ID procedure:	*`ch7312irq`*
Number of devices:	*`28`*
First vector number:	*`8`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Unchecked
IIP storage (bytes):	(unused)

Note that the standard Interrupt Identification Procedure *`ch7312fiq`* in board support module *`CSB7312.S`* assigns interrupt identification numbers 0 through 7 to eight FIQ interrupt sources which are then mapped them to AMX vectors 0 to 7. Interrupt Identification Procedure *`ch7312irq`* assigns interrupt identification numbers 0 through 27 to 28 IRQ interrupt sources which are then mapped them to AMX vectors 8 to 35. Since the EP7312 timers interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 12 and 13 respectively in the AMX Vector Table (IRQ base vector 8 plus timer interrupt identification numbers 4 and 5).

The Interrupt Identification Procedures *`ch7312fiq`* and *`ch7312irq`* do not support interrupt nesting.

You must compile clock source module *`CH7312T.C`*, assemble board support source module *`CSB7312.S`* and link the resulting object modules with the rest of your AMX application.

To use the AMX clock driver for the EP7312 general purpose timer, you must create a fast clock ISP root as described in Chapter 4.4. Use the following parameters in your definition of the clock ISP. Note that the interrupt from timer 1 is cleared by writing one word of any value to the timer end-of-interrupt register at device address *0x800006C0*. The interrupt from timer 2 is cleared by writing one word to device address *0x80000700*.

ISP Type:	Fast Clock Handler
ISP Root:	*ch7312clk*
Address #1:	*0x800006C0* for timer 1; *0x80000700* for timer 2
Value #1:	*0*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	32-bit

The board support module *CSB7312.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CH7312T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the EP7312 Clock Driver

If you wish to use a different EP7312 general purpose timer, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH7312T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or port the EP7312 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *CSB7312.S*.

The board support module *CSB7312.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

5.3.9 Freescale i.MX21 Clock Driver

The AMX clock driver for the Freescale i.MX21 General Purpose Timers is ready for use on the Freescale i.MX21ADS board. It is configured to use timer 1 operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *`CHIMX21T.C`*.

The i.MX21 timer 1 generates an IRQ interrupt which is assigned interrupt identification number 26. Board support module *`IMX21ADS.S`*, or your equivalent, must be used to service the i.MX21 interrupt controller. To use this clock driver on the Freescale i.MX21ADS board, you must select the IRQ interrupt model as described in Chapter 4.3 using the following parameters.

Use conforming IRQ interrupts	Checked
ID procedure:	*`chimx21irq`*
Number of devices:	*`64`*
First vector number:	*`0`*
Run-time vector checking:	Unchecked
Prioritized/nested by software:	Checked
IIP storage (bytes):	4
Use conforming FIQ interrupts	Unchecked

Note that the standard Interrupt Identification Procedure *`chimx21irq`* in board support module *`IMX21ADS.S`* assigns interrupt identification numbers 26, 25 and 24 to i.MX21 timers 0, 1 and 2 respectively. Since the i.MX21 timers are configured to interrupt via the IRQ interrupt exception, the interrupt identification numbers map to vector numbers 26, 25 and 24 in the AMX Vector Table (IRQ base vector 0 plus timer interrupt identification number).

The Interrupt Identification Procedure *`chimx21irq`* also supports interrupt nesting and requires 4 bytes of storage for this purpose.

You must compile clock source module *`CHIMX21T.C`*, assemble board support source module *`IMX21ADS.S`* and link the resulting object modules with the rest of your AMX application.

To use the AMX i.MX21 clock driver, you must create a clock ISP root as described in Chapter 4.3. Simply check the box next to the i.MX clock ISP on the list provided via the Prebuilt Clock ISPs... button.

The board support module *IMX21ADS.S* includes a board initialization procedure *chbrdinit* which must be called from your *main* program to initialize the interrupt system prior to launching AMX.

Clock driver module *CHIMX21T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the i.MX21 Clock Driver

If you wish to use a different i.MX21 general purpose timer, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CHIMX21T.C* and recompile the module. Edit instructions are included in the file.

If you wish to assign a different clock interrupt identification number or port the i.MX21 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *IMX21ADS.S*.

The board support module *IMX21ADS.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board.

Appendix A. Target Parameter File Specification

A.1 Target Parameter File Structure

The Target Parameter File is a text file structured as illustrated in Figure A.1-1. This file can be created and edited by the AMX Configuration Manager, a Windows® utility provided with AMX.

```
; AMX Target Parameter File
;
...LAUNCH         PERM,VNA
...HDW            PROC,VMASK,EVTROM,CACHE
...CACHE          FNCACHE,ICSIZE,ICPARAM,DCSIZE,DCPARAM
...DELAY          CPUFREQ
...HVALT          HVTABLE,HATABLE
...PIC            IRQ_FSIZE,FIQ_FSIZE
...FIQ            FVIDPROC,FVNBASE,FVNCOUNT,FVNCHECK
...IRQ            RVIDPROC,RVNBASE,RVNCOUNT,RVNCHECK
...NMI
;
;           Null Functions (optional; one line for each null function)
...NULLFN         FNNAME
;
;           Conforming ISP definitions (one line for each ISP)
...ISPA           ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
...ISPC           ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
;
;           Conforming fast clock ISP (no user code required)
...CLKFAST        CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST16      CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST32      CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
;           or one of the following conforming prebuilt clock ISPs:
...CLK21285             { 21285 Core Logic }
...CLKSA1100            { StrongARM SA-1100 }
...CLKAT91              { Atmel AT91 }
...CLK4510              { Samsung S3C4510 }
...CLK321               { XScale 80321 }
...CLKIMX               { Freescale i.MX1 or i.MX21 }
;           or conforming clock ISP (coded in assembly language)
...CLKA           CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
;           or conforming clock ISP (coded in C)
...CLKC           CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
;
;           AMX ROM Option (optional)
...ROMOPT         ROMADR,RAMADR
...ROMSM          ;Semaphore Manager
...ROMEM          ;Event Manager
...ROMMB          ;Mailbox Manager
...ROMMX          ;Message Exchange Manager
...ROMBM          ;Buffer Manager
...ROMMM          ;Memory Manager
...ROMCL          ;Circular List Manager
...ROMLL          ;Linked List Manager
...ROMTD          ;Time/Date Manager
```

Figure A.1-1 AMX Target Parameter File

The Target Parameter File consists of a sequence of directives consisting of a keyword of the form `...XXX` beginning in column one which is usually followed by a parameter list. Some directives require only a keyword with no parameters. Any line in the file which does not begin with a valid keyword is considered a comment and is ignored.

It is the purpose of this appendix to specify all AMX 4-ARM and AMX 4-Thumb directives by defining their keywords and the parameters, if any, which they require.

The example in Figure A.1-1 uses symbolic names for all of the parameters following each of the keywords. The symbol names in the Target Parameter File are replaced by the actual parameters needed in your system.

The order of keywords in the Target Parameter File is not critical. The order of the keywords in Figure A.1-1 may not match their order in the sample Target Parameter File provided with AMX.

It is expected that you will use the AMX Configuration Manager to create and edit your Target Parameter File. The Configuration Manager creates the directives using the parameters which you provide. Since these parameters are well described in Chapter 4, the parameter definitions presented in this appendix will be limited to the detail needed to form a working specification.

If you are unable to use AMX Configuration Manager utility, you should refer to the porting directions provided in Appendix A.3.

A.2 Target Parameter File Directives

The **AMX Launch Parameters** are defined as follows.

`...LAUNCH` `PERM,VNA`

`PERM`	0 if the AMX launch is temporary 1 if the AMX launch is permanent
`VNA`	0 if the ARM hard vector entries are to be alterable 1 if the ARM hard vector entries are NOT to be alterable

You must set `VNA` to 0 to allow AMX or your application to dynamically install exception handlers into the ARM hard vectors at run time. If you set `VNA` to 0, you must also set `EVTROM` to 0 in the `...HDW` keyword entry. If you set `VNA` to 1, you must initialize the ROM hard vectors for AMX use as described in Chapter 3.6.

The Target Parameter File includes a set of **hardware definitions**.

`...HDW` `PROC,VMASK,EVTROM,CACHE`

`PROC`	Processor identifier
`VMASK`	= `0xMM` = Exception Vector Mask
`EVTROM`	0 if the ARM hard vectors are to be in RAM 1 if the ARM hard vectors are to be in ROM
`CACHE`	0 if cache is to be ignored by AMX at launch 1 if cache is to be enabled by AMX at launch

The `PROC` parameter is a string used to identify the processor architecture. `PROC` must be one of:

`ARMV4`	ARM v4 architecture
`ARMV4T`	ARM v4T architecture (v4 with Thumb enhancements)
`ARMV5`	ARM v5 architecture
`ARMV5T`	ARM v5T architecture (v5 with Thumb enhancements)
`ARM7TDMI`	NEC ARM7TDMI processor (test chip)
`ARM710`	VLSI ARM710 processor
	ARM processors:
`ARM720T`, `ARM740T`, `ARM910T`, `ARM920T`, `ARM922T`, `ARM940T`, `ARM926EJ`, `ARM946E`, `ARM966E`, `ARM1020E`, `ARM1022E`, `ARM1026EJ`, `ARM1136J`, `ARM1156T2`, `ARM1176JZ`	
`SA110`	Intel (Digital Semiconductor) StrongARM® SA-110 processor
`SA1100`	Intel (Digital Semiconductor) StrongARM® SA-1100 processor
`SA1110`	Intel (Digital Semiconductor) StrongARM® SA-1110 processor
`80200`	Intel 80200 XScale™ processor
`80321`	Intel 80321 XScale™ processor
`AT91`	Atmel AT91 processor family
`EP73XX`	Cirrus Logic EP73xx processor family
`4510`	Samsung S3C4510 processor
`IMX1`	Freescale i.MX1 processor family
`IMX21`	Freescale i.MX21 processor family

If you are using a custom ASIC with an embedded ARM core, select the processor by choosing the ARM architecture which that core implements.

Note that AMX 4-Thumb only supports processors that are compliant with the ARM v4T or v5T architecture specification.

Set bit *N* of the *VMASK* Exception Vector Mask for each of the exceptions which are to be serviced by AMX and treated as fatal. For example, set this parameter to *0x38* to allow AMX to service the prefetch abort, data abort and address exceptions as fatal exceptions. This example leaves the reset, undefined instruction and SWI exceptions free for use by a debugger and leaves the IRQ and FIQ exceptions free for use by the AMX Interrupt Supervisor or your application. Bits in the mask are defined in Figure 3.1-1 but are interpreted as follows.

0x01	Reset and null pointer branches are fatal
0x02	Undefined Instruction is fatal
0x04	SWI software interrupt is fatal
0x08	Prefetch (Instruction) Abort is fatal
0x10	Data Abort is fatal
0x20	Address Exception is fatal
0x40	IRQ (Normal External Interrupt) is fatal
0x80	FIQ (Fast External Interrupt) is fatal

The *CACHE* parameter can be used to instruct AMX to enable the ARM instruction and data caches when AMX is launched. If the processor or architecture selected with parameter *PROC* has no cache control, set parameter *CACHE* to *0*.

Cache Override

The Target Parameter File includes a cache override definition.

```
...CACHE            FNCACHE,ICSIZE,ICPARAM,DCSIZE,DCPARAM
```

FNCACHE	Cache control function
ICSIZE	Instruction cache size (bytes)
ICPARAM	Instruction cache parameter (32-bit unsigned value)
DCSIZE	Data cache size (bytes)
DCPARAM	Data cache parameter (32-bit unsigned value)

The *...CACHE* directive allows you to customize the operation of the AMX cache control functions or to force the cache support functions *cjcfhwXcache* to call an alternate cache control function.

Parameter *FNCACHE* is the name of the cache control function. Parameters *ICSIZE*, *ICPARAM*, *DCSIZE* and *DCPARAM* are passed to the function as *unsigned long* C parameters.

Examples of the proper use of the *...CACHE* directive are provided in Appendix D.3.

To suppress AMX cache support, use the following form of the directive.

```
...CACHE            NOCACHE
```

Device I/O Delay

The Target Parameter File includes a device I/O delay definition.

```
...DELAY            CPUFREQ
```

CPUFREQ	ARM processor instruction execution frequency (MHz)

The *...DELAY* directive allows you to condition the delay loop of the AMX device I/O delay procedure *cjcfhwdelay* to match your hardware requirements. This directive allows AMX to use your estimate of the processor's instruction execution frequency defined by parameter *CPUFREQ* to derive the loop count needed to provide a one microsecond delay.

Alternate Hard Vector Table and Hard Address Table

The Target Parameter File can include an optional directive to define an alternate hard vector table and/or a hard address table.

```
...HVALT            HVTABLE,HATABLE
```

HVTABLE	Address of alternate hard vector table
HATABLE	Address of hard address table

By default, AMX assumes that the ARM hard vector table is located at memory address 0. You can use the *...HVALT* directive to identify for AMX where an alternate table is located. Parameter *HVTABLE* is the hexadecimal memory address of the alternate table. The numeric value must be expressed in a form acceptable to your assembler. The value 0 is acceptable and indicates that no alternate address is being provided.

By default, AMX uses a short branch in the hard vector table to dispatch to a particular AMX exception service procedure. To access procedures which reside more than +/-32Mb from the hard vector table, a hard address table must be provided. The hard address table is a block of 32 bytes of long-aligned, alterable memory which must reside within +4100 or -4092 bytes of the base of the hard vector table. AMX installs pointers to its exception service procedures into this array and uses long branches to dispatch to them. Parameter *HATABLE* is the hexadecimal memory address of the hard address table. The numeric value must be expressed in a form acceptable to your assembler. If the parameter is 0, then long branches will not be supported by AMX.

Interrupt Nesting Support

The Target Parameter File can include an optional directive to enable support for nested interrupts using a software programmable interrupt controller.

```
...PIC            IRQ_FSIZE,FIQ_FSIZE
```

IRQ_FSIZE	Frame size required by IRQ Interrupt Identification Procedure
FIQ_FSIZE	Frame size required by FIQ Interrupt Identification Procedure

The *...PIC* directive indicates that the Interrupt Identification Procedure (IIP) for the IRQ and/or FIQ interrupt exception supports interrupt nesting. Instructions for creating such an IIP are provided in Appendix E.4. The AMX Interrupt Supervisor will allocate *IRQ_FSIZE* (*FIQ_FSIZE*) bytes of history storage on its Interrupt Stack for use by the IRQ (FIQ) IIP. The numeric value must be expressed in a form acceptable to your assembler.

If the *...PIC* directive is omitted, neither the IRQ or FIQ Interrupt Identification Procedure (IIP) can support nesting. If the *...PIC* directive is used and either *IRQ_FSIZE* or *FIQ_FSIZE* is 0, the corresponding IIP cannot support nesting.

Most of the Interrupt Identification Procedures in the board support modules provided with AMX support interrupt nesting. If you use one of these IIPs, you must allocate 4 bytes for its use by including one of the following directives in your Target Parameter File.

```
      ...PIC 4,0        ; AMX IRQ interrupt model
or    ...PIC 0,4        ; AMX FIQ interrupt model
or    ...PIC 4,4        ; AMX full interrupt model
```

AMX Interrupt Model

The Target Parameter File defines the interrupt model which determines how the ARM FIQ and IRQ interrupt exceptions are to be used in your AMX application. The `...FIQ` and `...IRQ` directives are used for this purpose. The parameter descriptions can be found following the interrupt model summary.

Full Interrupt Model

```
...FIQ          FVIDPROC,FVNBASE,FVNCOUNT,FVNCHECK
...IRQ          RVIDPROC,RVNBASE,RVNCOUNT,RVNCHECK
```

Do not set the FIQ or IRQ mask bits in parameter *VMASK* in the *...HDW* directive.
Do not use the *...NMI* directive.

The FIQ and IRQ interrupt exceptions are both serviced by the AMX Interrupt Supervisor. All interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) both the *F* and *I* bits in the *CPSR*.

IRQ Interrupt Model

```
...IRQ          RVIDPROC,RVNBASE,RVNCOUNT,RVNCHECK
```

Do not set the IRQ mask bit in parameter *VMASK* in the *...HDW* directive.
Do not use the *...FIQ* directive.

The IRQ interrupt exception is serviced by the AMX Interrupt Supervisor. All IRQ interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) the *I* bit in the *CPSR*. The *F* bit is not altered.

If the FIQ interrupt is not used for any purpose, you can set the FIQ mask bit in parameter *VMASK* in the *...HDW* directive to force AMX to treat a spurious FIQ interrupt as a fatal exception.

FIQ Interrupt Model

```
...FIQ          FVIDPROC,FVNBASE,FVNCOUNT,FVNCHECK
```

Do not set the FIQ mask bit in parameter *VMASK* in the *...HDW* directive.
Set the IRQ mask bit in parameter *VMASK* in the *...HDW* directive.
Do not use the IRQ interrupt for any purpose.
Do not use the *...IRQ* or *...NMI* directives.

The FIQ interrupt exception is serviced by the AMX Interrupt Supervisor. All FIQ interrupts are serviced using conforming ISPs. AMX disables (enables) interrupts by setting (clearing) the *F* bit in the *CPSR*. The *I* bit is set to unconditionally inhibit IRQ interrupts.

Null Interrupt Model

Omit the `...FIQ` and `...IRQ` directives.

If you use the FIQ (or IRQ) interrupt exception for your own nonconforming purpose, do not set the corresponding FIQ (or IRQ) mask bit in parameter *`VMASK`* in the `...HDW` directive.

If the FIQ (or IRQ) interrupt is not used for any purpose, you can set the FIQ (or IRQ) mask bit in parameter *`VMASK`* in the `...HDW` directive to force AMX to treat a spurious FIQ (or IRQ) interrupt as a fatal exception.

This model is rarely used since no AMX clock or other AMX interrupt devices are supported.

Since **multiple devices** can generate interrupts through the FIQ and IRQ interrupt exception vectors, the exception definitions are as follows.

```
...FIQ              FVIDPROC,FVNBASE,FVNCOUNT,FVNCHECK
...IRQ              RVIDPROC,RVNBASE,RVNCOUNT,RVNCHECK
```

xVIDPROC	Name of the Interrupt Identification Procedure
xVNBASE	Base vector number in the AMX Vector Table
xVNCOUNT	Number of vectors, beginning at *xVNBASE*, required by devices interrupting through the exception vector
xVNCHECK	*0* if run-time vector number checking is disabled *1* if run-time vector number checking is enabled

In the discussion which follows, the leading *x* represents *F* for fast FIQ interrupts or *R* for normal IRQ interrupts.

If more than one device can cause the interrupt exception to occur, you must provide an Interrupt Identification Procedure (see Chapter 3.2). Parameter *xVIDPROC* is the name of the procedure. Note that separate procedures are required for the FIQ and IRQ interrupts if both are serviced by the AMX Interrupt Supervisor. The procedure returns a number from *0* to *n-1* identifying which of the *n* devices generated the interrupt currently under service. Sample Interrupt Identification Procedures can be found in the board support modules provided with AMX.

An entry in the AMX Vector Table is reserved for every device serviced by AMX through the interrupt exceptions. Parameter *xVNBASE* defines the base vector number assigned by you to the devices attached to the particular interrupt exception. The interrupt identification number provided by Interrupt Identification Procedure *xVIDPROC* is added to *xVNBASE* to derive the AMX vector number for the device.

Parameter *xVNCOUNT* defines the number of AMX vectors starting at vector number *xVNBASE* which are needed to accommodate the devices attached to the particular interrupt exception. The AMX Vector Table will contain *NVEC* entries where *NVEC* is the maximum of *FVNBASE+FVNCOUNT* and *RVNBASE+RVNCOUNT*.

The AMX Interrupt Supervisor can check that the derived vector number lies within the range *xVNBASE* to *xVNBASE+xVNCOUNT-1* in the AMX Vector Table. If it does not, AMX calls the Fatal Exception Handler indicating that an unidentified interrupt exception was detected via hard vector *CJ_PRVNFIQ* or *CJ_PRVNIRQ*.

To reduce interrupt service overhead, vector number validation can be disabled. Set parameter *xVNCHECK* to *0/1* to disable/enable vector number checking by the AMX Interrupt Supervisor.

If a **single device** generates an interrupt through the FIQ or IRQ interrupt exception vectors, the exception definition is as follows. The Interrupt Identification Procedure is omitted and the vector count is set to 1. Do not omit the leading comma.

```
...FIQ              ,FVNBASE,1,0
...IRQ              ,RVNBASE,1,0
```

Pseudo Non-Maskable Interrupt (NMI)

The Target Parameter File can include an optional directive to permit the FIQ interrupt exception to be used as a pseudo non-maskable interrupt (see Chapter 3.3).

```
...NMI
```

If your AMX application uses the IRQ or null interrupt model, the FIQ interrupt exception will not be used by the AMX Interrupt Supervisor. If you wish to use the FIQ interrupt exception as a pseudo non-maskable interrupt, add the `...NMI` directive to your Target Parameter File.

When using the FIQ interrupt exception as a pseudo non-maskable interrupt, do not set the corresponding FIQ mask bit in parameter *VMASK* in the `...HDW` directive.

Null Function Declarations

To create a null function, a function that does nothing, include the following directive in your Target Parameter File.

```
...NULLFN            FNNAME
```

FNNAME	Name given to the null function

For every `...NULLFN` directive, your Target Configuration Module will include a public assembly language function with name given by your parameter *FNNAME*. The function will do nothing but return to the caller.

Conforming ISP Declarations

The Target Parameter File must include a definition of an ISP root for each conforming Interrupt Service Procedure (ISP) which you intend to use in your application. The ISP root definition is provided using the one of the following directives. The ISP root is declared using `...ISPC` if its Interrupt Handler is coded in C or `...ISPA` if its Interrupt Handler is coded in assembly language.

```
...ISPC          ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
...ISPA          ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
```

ISPROOT	Name of the ISP root entry point
HANDLER	Name of the public device Interrupt Handler
VNUM	AMX vector number assigned to the device
PARAM	Interrupt Handler parameter
PARTYPE	Parameter *PARAM* type

If your Interrupt Handler does not require a parameter, leave field *PARAM* blank (empty) and set *PARTYPE* to 0.

If your Interrupt Handler requires a numeric parameter, set *PARAM* to the 32-bit signed or unsigned value and set *PARTYPE* to 0. The numeric value must be expressed in a form acceptable to your assembler.

If your Interrupt Handler requires a pointer to a public variable as a parameter, let *PARAM* be the name of that variable and set *PARTYPE* to 1.

VNUM defines the AMX vector number which you have assigned to the device. If the device interrupts via the FIQ interrupt, *VNUM* will be between *FVNBASE* and *FVNBASE+FVNCOUNT-1* (see directive *...FIQ*). If the device interrupts via the IRQ interrupt, *VNUM* will be between *RVNBASE* and *RVNBASE+RVNCOUNT-1* (see directive *...IRQ*).

If *VNUM* is greater than or equal to *0*, AMX will automatically install the pointer to the ISP root *ISPROOT* into vector number *VNUM* in the AMX Vector Table when AMX is launched. The pointer will be installed by AMX before any application Restart Procedures execute. Consequently, you must ensure that interrupts from the device are not possible at the time AMX is launched.

If *VNUM* is *-1*, you must provide a Restart Procedure or task which installs the pointer to the ISP root *ISPROOT* into the AMX Vector Table using AMX procedure `cjksivtwr` or `cjksivtx`.

Note

Parameter *VNUM* cannot be adjusted using the AMX Configuration Builder. This parameter is provided for compatibility with other AMX implementations.

AMX Clock Handler Declaration

The Target Parameter File must include a definition of an ISP root for your AMX clock handler. The clock ISP root definition must be provided using one of the following directives. The clock ISP root is declared using *...CLKC* if its Interrupt Handler is coded in C or *...CLKA* if its Interrupt Handler is coded in assembly language. The clock ISP root can be declared using *...CLKFAST* if an Interrupt Handler is not required to service the clock.

...CLK21285	Prebuilt 21285 Core Logic Clock ISP
...CLKSA1100	Prebuilt SA-1100 Clock ISP
...CLKAT91	Prebuilt Atmel AT91 Clock ISP
...CLK4510	Prebuilt Samsung S3C4510 Clock ISP
...CLK321	Prebuilt XScale 80321 Clock ISP
...CLKIMX	Prebuilt Freescale i.MX Clock ISP
...CLKC	*CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE*
...CLKA	*CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE*
...CLKFAST	*CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM*
...CLKFAST16	*CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM*
...CLKFAST32	*CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM*

CLKROOT	Name of the clock ISP root entry point
CLKHAND	Name of the public clock device Interrupt Handler
VNUM	AMX vector number assigned to the clock device
PARAM	Interrupt Handler parameter
PARTYPE	Parameter *PARAM* type

If your clock Interrupt Handler does not require a parameter, leave field *PARAM* blank (empty) and set *PARTYPE* to 0.

If your clock Interrupt Handler requires a numeric parameter, set *PARAM* to the 32-bit signed or unsigned value and set *PARTYPE* to 0. The numeric value must be expressed in a form acceptable to your assembler.

If your clock Interrupt Handler requires a pointer to a public variable as a parameter, let *PARAM* be the name of that variable and set *PARTYPE* to 1.

The definition of parameter *VNUM* is exactly the same as that described for conforming ISPs declared using the *...ISPC* or *...ISPA* directives. However, unless warranted by exceptional circumstances, parameter *VNUM* should always be set to *-1* in the declaration of your clock ISP root. It is the responsibility of your clock initialization procedure to install the pointer to the ISP root *ISPROOT* into the AMX Vector Table.

Note

Parameter *VNUM* cannot be adjusted using the AMX Configuration Builder. This parameter is provided for compatibility with other AMX implementations.

If your clock can be serviced by writing one or two *n*-bit values to a device I/O port, you can use the `...CLKFAST` directive to create a very fast clock ISP root with no application code required. The general form of the `...CLKFAST` directive is as follows.

```
...CLKFAST         CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
```

CLKROOT	Name of the clock ISP root entry point
CLKADR	32-bit numeric device memory address
CLKCMD	8-bit numeric command
CLKADR2	32-bit numeric secondary device memory address
CLKCMD2	8-bit numeric secondary command
IODELAY	Delay (µs) required between I/O commands
VNUM	AMX vector number assigned to the clock device

The numeric parameters must be expressed in a form acceptable to your assembler. Parameters *CLKADR2*, *CLKCMD2*, *IODELAY* and *VNUM* can be omitted if they are not required. If a parameter is omitted, its field must be left blank (empty) and the comma to the left of the field must be retained. If the resulting `...CLKFAST` directive ends with a string of commas because the intervening parameters have all been omitted, it is acceptable to delete the trailing commas.

The clock ISP root will dismiss the clock interrupt by writing the 8-bit value *CLKCMD* to the 32-bit device memory address *CLKADR*. If parameter *CLKADR2* is present in the `...CLKFAST` directive, the clock ISP root will then write the 8-bit value to the 32-bit device memory address *CLKADR2*. If parameter *CLKADR2* is present, parameter *CLKCMD2* must also be present. If this second device I/O command is not required, leave both *CLKCMD2* and *CLKADR2* blank (empty).

If two I/O commands are provided, parameter *IODELAY* can be used to define the delay, if any, required after the first command before the second command can be issued. The delay is provided by a call to AMX procedure *cjcfhwdelay* (see directive `...DELAY`).

If there is no need for a delay or a second command is not required, leave the *IODELAY* field blank (empty).

Parameter *VNUM* has been described on the preceding page. If parameter *VNUM* is omitted, then a value of *-1* is assumed for *VNUM*.

Use the `...CLKFAST16` directive if 16-bit values must be written to the clock.
Use the `...CLKFAST32` directive if 32-bit values must be written to the clock.

AMX ROM Option

To use the AMX ROM option, the Target Parameter File must include the following directives.

```
...ROMOPT          ROMADR,RAMADR
...ROMSM           ;Semaphore Manager
...ROMEM           ;Event Manager
...ROMMB           ;Mailbox Manager
...ROMMX           ;Message Exchange Manager
...ROMBM           ;Buffer Manager
...ROMMM           ;Memory Manager
...ROMCL           ;Circular List Manager
...ROMLL           ;Linked List Manager
...ROMTD           ;Time/Date Manager
```

Parameter *ROMADR* is the absolute physical ROM address at which the AMX ROM image is to be located.

Parameter *RAMADR* is the absolute physical RAM address of a block of 32 bytes reserved for use by AMX.

Both *ROMADR* and *RAMADR* must specify memory addresses which are long aligned.

Parameters *ROMADR* and *RAMADR* must be expressed as undecorated hexadecimal numbers. An undecorated hexadecimal number is a hexadecimal number expressed without the leading or trailing symbols used by programming languages to identify such numbers.

Language	**Hexadecimal**	**Undecorated**
C	*0xABCDEF01*	*ABCDEF01*
Assembler (Intel)	*0ABCDEF01H*	*ABCDEF01*
Assembler (Motorola)	*$ABCDEF01*	*ABCDEF01*

Keywords *...ROMxx* are used to identify the AMX managers which you wish to commit to the AMX ROM. If you do not want a particular manager to be in the ROM, omit the corresponding keyword statement from the Target Parameter File or insert the comment character *;* in front of the keyword.

A.3 Porting the Target Parameter File

It is expected that you will use the AMX Configuration Manager to create and edit your Target Parameter File. If you are unable to use the AMX Configuration Manager utility, you will have to create and edit your Target Parameter File using a text editor.

You should begin by choosing one of the sample Target Parameter Files provided with AMX. Choose the Target Parameter File for the Sample Program which operates on the evaluation board which most closely matches your target hardware. Edit the parameters in all directives to meet your requirements. Follow the specifications provided in Appendix A.2 and adhere to the detailed parameter definitions given in the presentation of the AMX Configuration Manager screens in Chapter 4.

The AMX Configuration Manager includes its own copy of the AMX Configuration Generator which it uses to produce your Target Configuration Module from the Target Configuration Template File and the directives in your Target Parameter File. If you are unable to use the Configuration Manager, you will have to use the stand alone version of the AMX Configuration Generator.

The command line required to run the Configuration Generator and use it to produce a Target Configuration Module *HDWCFG.S* from the AMX 4-ARM Target Configuration Template File *CJ402HDW.CT* and a Target Parameter File called *HDWCFG.UP* is as follows:

```
CJ402CG HDWCFG.UP CJ402HDW.CT HDWCFG.S
```

The command line required to run the Configuration Generator and use it to produce a Target Configuration Module *HDWCFG.S* from the AMX 4-Thumb Target Configuration Template File *CJ422HDW.CT* and a Target Parameter File called *HDWCFG.UP* is as follows:

```
CJ422CG HDWCFG.UP CJ422HDW.CT HDWCFG.S
```

If you are not doing your development on a PC or compatible, you may still be able to port the Configuration Generator to your development system as described in Appendix C of the AMX User's Guide.

This page left blank intentionally.

Appendix B. AMX for ARM Service Procedures

B.1 Summary of Services

AMX 4-ARM and AMX 4-Thumb provide a collection of target dependent AMX service procedures for use with the ARM processor and compatibles and the C compilers which support them. These procedures are summarized below.

Interrupt Control (class *ksi*)

cjksivtp	Fetch pointer to the AMX Vector Table
cjksivtrd	Read an entry from the AMX Vector Table
cjksivtwr	Write an entry into the AMX Vector Table
cjksivtx	Exchange an entry in the AMX Vector Table

Processor and C Interface Procedures (class *cf*)

In addition to the services provided by AMX and its managers, the AMX Library includes several C procedures of a general nature which simplify application programming in real-time systems on your target processor.

cjcfccsetup	Setup C environment
cjcfdi	Disable interrupts
cjcfdiprev	Disable interrupts; return previous *CPSR*
cjcfei	Enable interrupts
cjcfflagrd	Read the current processor status register (*CPSR*)
cjcfflagwr	Write to the current processor status register (*CPSR*)
cjcfhvrd	Read hard vector and decode branch address
cjcfhvwr	Write branch instruction into hard vector
cjcfhwdelay	Delay *n* microseconds
cjcfhwbcache	Enable/disable the instruction and data cache
cjcfhwdcache	Enable/disable the data cache
cjcfhwicache	Enable/disable the instruction cache
cjcfhwbflush	Flush the instruction and data cache
cjcfhwdflush	Flush the data cache
cjcfhwiflush	Flush the instruction cache
cjcfin8	Read an 8-bit input port
cjcfin16	Read a 16-bit input port
cjcfin32	Read a 32-bit input port
cjcfjlong	Long jump to a mark set by *cjcfjset*
cjcfjset	Set a mark for a subsequent long jump by *cjcfjlong*
cjcfmcopy	Copy a block of memory
cjcfmodcpsr	Read/modify the Current Program Status Register (*CPSR*)
cjcfmset	Set (fill) a block of memory
cjcfout8	Write an 8-bit value to an output port
cjcfout16	Write a 16-bit value to an output port
cjcfout32	Write a 32-bit value to an output port
cjcfstkjmp	Switch stacks and jump to a new procedure
cjcftag	Convert a string to an AMX tag value

Processor and C Interface Procedures (class *cf*) (continued...)

cjcfvol8	Read a volatile 8-bit variable
cjcfvol16	Read a volatile 16-bit variable
cjcfvol32	Read a volatile 32-bit variable
cjcfvolpntr	Read a volatile pointer variable

The AMX Library also includes several C procedures which are used privately by KADAK. These procedures, although available for your use, are not documented in this manual and are subject to change at any time. The procedures are briefly described in source file *CJZZZUB.S*. Prototypes will be found in file *CJZZZIF.H*. The register array structures *cjxregs* and *cjxmregs* which they use are defined in file *CJZZZKT.H*.

cjcfmregld	Load ARM banked registers from an extended register array
cjcfmregst	Store ARM banked registers into an extended register array
cjcfregld	Load ARM registers from a register array
cjcfregst	Store ARM registers into a register array
cjcfsint	Generate a software initiated exception

B.2 Service Procedures

A description of all processor dependent AMX 4-ARM and AMX 4-Thumb service procedures is provided in this appendix. The descriptions are ordered alphabetically for easy reference.

Italics are used to distinguish programming examples. Procedure names and variable names which appear in narrative text are also displayed in italics. Occasionally a lower case procedure name or variable name may appear capitalized if it occurs as the first word in a sentence.

Vertical ellipses are used in program examples to indicate that a portion of the program code is missing. Most frequently this will occur in examples where fragments of application dependent code are missing.

```
:
: /* Dismiss device interrupt */
:
```

Capitals are used for all defined AMX filenames, constants and error codes. All AMX procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3 of the AMX User's Guide.

A consistent style has been adopted for each description. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

Purpose A one-line statement of purpose is always provided.

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure □ Exit Procedure

This block is used to indicate which of your AMX application procedures can call the AMX procedure. The term ISP refers to the Interrupt Handler of a conforming ISP. A filled in box indicates that the procedure is allowed to call the AMX procedure. In the above example, only tasks and Restart Procedures would be allowed to call the procedure.

Setup The prototype of the AMX procedure is shown.
The AMX header file in which the prototype is located is identified.
Include AMX header file *CJZZZ.H* for compilation.

File *CJZZZ.H* is a generic AMX include file which automatically includes the correct subset of the AMX header files for a particular target processor. If you include *CJZZZ.H* instead of its KADAK part numbered counterpart (*CJnnn.H*), your AMX application source modules will be readily portable to other processors without editing.

Description Defines all input parameters to the procedure and expands upon the purpose or method if required.

Interrupts AMX procedures frequently must deal with the processor interrupt mask. The effect of each AMX procedure on the interrupt state is defined according to the following legend.

■ Disabled ■ Enabled (Not in ISP) ■ Restored

D	E	R	Effect on Interrupts
□	□	□	Untouched
■	□	□	Disabled and left disabled upon return
□	■	□	Enabled and left enabled upon return
■	■	□	Disabled and then enabled upon return
■	□	■	Disabled and then, prior to return, restored to the state in effect upon entry to the procedure
■	■	■	Disabled, possibly briefly enabled and then, prior to return, restored to the state in effect upon entry to the procedure

The warning (Not in ISP) will be present as a reminder that when the Interrupt Handler of a conforming ISP calls the AMX procedure, interrupts will NOT be explicitly enabled by the AMX procedure. If interrupts are enabled when an Interrupt Handler calls the AMX procedure, they will be enabled upon return.

Returns The outputs, if any, produced by the procedure are always defined.

Most AMX procedures return an integer error status identified as a *`CJ_ERRST`*. Note that *`CJ_ERRST`* is not a C data type. *`CJ_ERRST`* is defined (using *`#define`*) to be an *`int`* allowing error codes to be easily handled as integers but readily identified as AMX error codes.

Restrictions If any restrictions on the use of the procedure exist, they are described.

Note Special notes, suggestions or warnings are offered where necessary.

Task Switch Task switching effects, if any, are described.

Example An example is provided for each of the more complex AMX procedures. The examples are kept simple and are intended only to illustrate the correct calling sequence.

See Also A cross reference to other related AMX procedures is always provided if applicable.

Purpose **Setup C Environment**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfccsetup(void);
```

Description Use *cjcfccsetup* to setup all low level processor registers to meet the requirements of a particular C compiler. For example, the C compiler may assume that some data variables can be accessed using a particular register which always points to the data. However, when mixing languages, you may find that when a C procedure is called from assembly language, the register assumptions are not valid. A call to *cjcfccsetup* on entry to the C procedure will setup the correct register content.

Interrupts □ Disabled □ Enabled □ Restored

Returns The registers, if any, which are required by C are set to the values which they contained when AMX was launched.

Restrictions Use *cjcfccsetup* with care. You may inadvertently cause a register to be set which violates the register preservation rules of the other language.

Purpose **Disable or Enable Interrupts**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototypes for *cjcfXi* are in file *CJZZZTF.H*.
Prototype for *cjcfdiprev* is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfdi(void);
CJ_TYFLAGS CJ_CCPP cjcfdiprev(void);
void CJ_CCPP cjcfei(void);
```

Description Use *cjcfdi* (or *cjcfdiprev*) to briefly disable all sources of interrupt. Immediately thereafter use *cjcfei* to enable all sources of interrupt again.

Interrupts Disabled by *cjcfdi* and *cjcfdiprev*
Enabled by *cjcfei*

Returns *Cjcfdi* and *cjcfei* return nothing.
Cjcfdiprev returns the previous content of the Current Processor Status Register (*CPSR*).

The *I* and *F* bits in the Current Processor Status Register are set to *1* to disable interrupts or set to *0* to enable interrupts.

Restrictions ISPs must not use *cjcfei*. If nested interrupts are supported in your application, ISPs must always use *cjcfflagwr* to restore interrupts to the state determined by an earlier call to *cjcfdiprev* or *cjcfflagrd*.

Interrupts should be enabled within a short time after they are disabled or system performance will be degraded.

If the IRQ or FIQ exception is not used for device interrupts which are serviced through the AMX Interrupt Supervisor, these functions will not alter the corresponding *I* or *F* bit in the *CPSR*.

These procedures can also be used to manipulate the Current Processor Status Register (*CPSR*) prior to launching AMX or after exiting from AMX. Therefore, you can call these procedures from your *main()* program before or after your call to *cjkslaunch()*. Of course, your *main()* program must be executing in a privileged mode as required by AMX.

See Also *cjcfflagrd, cjcfflagwr, cjcfmodcpsr*

Purpose **Read or Write Processor Flags**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototypes in file `CJZZZIF.H` or macros in file `CJZZZCC.H`.

```
#include "CJZZZ.H"
CJ_TYFLAGS CJ_CCPP cjcfflagrd(void);
void CJ_CCPP cjcfflagwr(CJ_TYFLAGS flags);
```

Description *Cjcfflagrd* returns the actual state of the Current Processor Status Register (*CPSR*).

Cjcfflagwr updates the Current Processor Status Register (*CPSR*) by writing the parameter *flags* directly to the register.

Interrupts □ Untouched by *cjcfflagrd* ■ Restored by *cjcfflagwr*

Returns *Cjcfflagrd* returns the actual state of the Current Processor Status Register (*CPSR*).
Cjcfflagwr returns nothing.

Restrictions These procedures can also be used to manipulate the Current Processor Status Register (*CPSR*) prior to launching AMX or after exiting from AMX. Therefore, you can call these procedures from your *main()* program before or after your call to *cjkslaunch()*. Of course, your *main()* program must be executing in a privileged mode as required by AMX.

Warning

Do not use *cjcfflagwr()* to switch operating modes. If you must switch to another operating mode for some reason, use *cjcfmodcpsr()* to do so.

See Also *cjcfdi, cjcfdiprev, cjcfei, cjcfmodcpsr*

Purpose **Read Hard Vector and Decode Branch Address**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjcfhvrd(int hvector, CJ_ISPPROC *oldproc);
```

Description *hvector* is the ARM hard vector address. The hard vectors can be referenced using the exception vector mnemonics (*CJ_PRVNxxx*) defined in AMX 4-ARM header file *CJ402KT.H* or AMX 4-Thumb header file *CJ422KT.H*.

oldproc is a pointer to storage for the address of the exception service procedure to which the hard vector branches when the exception occurs. Procedure *cjcfhvrd* decodes the branch instruction stored at the hard vector memory location to determine the address of the exception service procedure.

Interrupts □ Disabled □ Enabled □ Restored

Returns Error status is returned.

CJ_EROK	Call successful.

**oldproc* contains the address of the exception service procedure.

Errors returned:

For all errors, **oldproc* is undefined on return.

CJ_ERRANGE	Invalid hard vector address *hvector*.
CJ_ERFORMAT	No branch instruction installed at address *hvector*.

See Also *cjcfhvwr*

Purpose **Write Branch Instruction into Hard Vector**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjcfhvwr(int hvector, CJ_ISPPROC newproc);
```

Description *hvector* is the ARM hard vector address. The hard vectors can be referenced using the exception vector mnemonics (*CJ_PRVNxxx*) defined in AMX 4-ARM header file *CJ402KT.H* or AMX 4-Thumb header file *CJ422KT.H*.

newproc is a pointer to the exception service procedure to which you wish the hard vector to branch when the exception occurs. AMX will install a "branch to *newproc*" instruction into the hard vector memory location.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful.

Errors returned:

CJ_ERNOACCESS	Hard Vector Table is not alterable.
CJ_ERRANGE	Invalid hard vector address *hvector*.
CJ_ERFORMAT	Cannot reach *newproc* with a branch instruction.

Restrictions If the ARM processor targeted by your Target Parameter File has an instruction and/or data cache, AMX will flush the data cache and invalidate the instruction cache after writing the branch instruction into the hard vector.

If you configured AMX to ignore the caches, AMX will not touch the cache after installing the branch instruction into the hard vector. If the caches are actually enabled, you may experience a malfunction when the exception finally occurs.

See Also *cjcfhvrd*

Purpose	**Delay n Microseconds**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfhwdelay(int n);
```

Description *n* is the delay interval measured in microseconds.

Use *cjcfhwdelay* to generate a software delay loop of approximately *n* microseconds. This procedure is intended for use in device drivers which must introduce device access delays to avoid violating the minimum timing delay needed between sequential references to a device I/O port.

The *...DELAY* directive in your Target Parameter File is used by AMX to derive the delay loop count needed to produce an *n* microsecond delay.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Nothing

Note This procedure can be used at any time, even prior to launching AMX or after exiting from AMX.

If the *...DELAY* directive in your Target Parameter File indicates that the processor frequency is *0*, then you must install the frequency value into the public *long* variable *cjcfhwdelayf* prior to launching AMX. If you call procedure *cjcfhwdelay()* prior to launching AMX, be sure that variable *cjcfhwdelayf* is initialized before making the call.

Purpose **Flush and Enable/Disable Caches**

Used by ■ Task ☐ ISP ☐ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfhwbcache(int operation);
void CJ_CCPP cjcfhwdcache(int operation);
void CJ_CCPP cjcfhwicache(int operation);
```

Description *operation* = *0* to force the caches to be flushed and disabled.

operation = *1* to force the caches to be flushed and enabled.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Nothing

Cjcfhwbcache flushes and disables (or enables) both the data and instruction caches.

Cjcfhwdcache flushes and disables (or enables) only the data cache.

Cjcfhwicache flushes and disables (or enables) only the instruction cache.

Note These procedures can be called even if your Target Parameter File indicates that you are targeting an ARM processor with no cache or only one kind of cache. In such cases, the procedures only affect the caches which exist.

Appendix D is recommended reading for anyone wishing to manipulate the caches.

Restrictions These procedures do not disable interrupts. You MUST disable interrupts prior to disabling or enabling the cache. Use *cjcfdiprev* to disable interrupts and *cjcfflagwr* to restore interrupts.

If you call these procedures from your *main()* program before or after your call to *cjkslaunch()*, you must use *cjcfflagrd* and *cjcfflagwr* to manipulate the *CPSR* to enable and restore interrupts.

Use caution when calling these procedures or system performance will be degraded, especially if the cache sizes are large.

cjcfhwbflush cjcfhwdflush cjcfhwiflush

Purpose **Flush (Invalidate) Caches**

Used by ■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototypes are in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfhwbflush(void *cpntr, CJ_T32U csize);
void CJ_CCPP cjcfhwdflush(void *cpntr, CJ_T32U csize);
void CJ_CCPP cjcfhwiflush(void *cpntr, CJ_T32U csize);
```

Description *cpntr* is a pointer to the block of data (or instruction) memory which is to be flushed. The region of the data (or instruction) cache to which this block of memory is mapped will be flushed to memory and invalidated.

csize is the size of the memory block which is to be flushed. *Csize* must be a multiple of *4*. *Csize* must be *>0*. *Csize* bytes in the data (or instruction) cache will be invalidated.

Interrupts □ Disabled □ Enabled □ Restored

Returns Nothing

Cjcfhwbflush flushes both the data cache and instruction cache.

Cjcfhwdflush flushes only the data cache.

Cjcfhwiflush flushes only the instruction cache.

These procedures flush and invalidate the instruction/data caches for the specified memory range.

Note These procedures can be called even if your Target Parameter File indicates that you are targeting an ARM processor with no cache or only one kind of cache. In such cases, the procedures only affect the caches which exist.

Restrictions These procedures do not disable interrupts. It is recommended that you disable interrupts prior to flushing the cache. Use *cjcfdiprev* to disable interrupts and *cjcfflagwr* to restore interrupts.

If you call these procedures from your *main()* program before or after your call to *cjkslaunch()*, you must use *cjcfflagrd* and *cjcfflagwr* to manipulate the *CPSR* to enable and restore interrupts.

Use caution when calling these procedures or system performance will be degraded, especially if the flush size *csize* is large. In particular, ISP Handlers and Timer Procedures should not use these procedures.

cjcfin8 cjcfin16 cjcfin32

Purpose **Read an 8, 16 or 32-Bit Input Port**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.

```
#include "CJZZZ.H"
CJ_T8 CJ_CCPP cjcfin8(void *port);
CJ_T16 CJ_CCPP cjcfin16(void *port);
CJ_T32 CJ_CCPP cjcfin32(void *port);
```

Description *port* is the address of an 8, 16 or 32-bit memory-mapped device input port.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns *Cjcfin8* returns an 8-bit signed value.
Cjcfin16 returns a 16-bit signed value.
Cjcfin32 returns a 32-bit signed value.

Example

```
#include "CJZZZ.H"

                             /* Console status register   */
#define CONSTAT ((CJ_T8 *)0xFFFA002DL)
                             /* Console data register     */
#define CONDATA ((CJ_T8 *)0xFFFA002FL)


void CJ_CCPP conout(char ch) {

                             /* Wait for ready            */
  while ( (cjcfin8(CONSTAT) & 0x80) == 0 )
        ;

                             /* Write character           */
  cjcfout8(CONDATA, (CJ_T32)ch);
  }
```

See Also *cjcfout8, cjcfout16, cjcfout32*

Purpose **cjcfjset Sets a Mark for a Long Jump**
cjcfjlong Long Jumps to that Mark

These procedures are provided for AMX portability. They are not replacements for C library procedures *longjmp* or *setjmp* although they function in a similar manner.

Used by ■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZTF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfjlong(struct cjxjbuf *jbuf, int value);
int CJ_CCPP cjcfjset(struct cjxjbuf *jbuf);
```

Description *jbuf* is a pointer to a jump buffer to be used to mark the processor state at the time *cjcfjset* is called and to restore that state when *cjcfjlong* is subsequently called.

The processor dependent structure *cjxjbuf* is defined in file *CJZZZCC.H*.

value is an integer value to be returned to the *cjcfjset* caller when *cjcfjlong* initiates the long jump return. *Value* cannot be 0. If *value = 0*, *cjcfjlong* will replace it with *value = 1*.

Interrupts □ Disabled □ Enabled □ Restored

Returns *Cjcfjset* returns 0 when initially called to establish the mark. *Cjcfjset* returns *value* (non 0) when *cjcfjlong* is called to do the long jump to the mark established by the initial *cjcfjset* call.

There is no return from *cjcfjlong*.

Restrictions *Cjcfjset* must be called prior to any call to *cjcfjlong*. Each call must reference the same jump buffer. The jump buffer must remain unaltered between the initial *cjcfjset* call and the subsequent *cjcfjlong* long jump return.

Under no circumstances should one task attempt a long jump using a jump buffer set by another task.

Example

```
#include "CJZZZ.H"

void CJ_CCPP dowork(struct cjxjbuf *jbp);

static struct cjxjbuf jumpbuffer;

#define STACKSIZE 512        /* Stack size (longs)         */
#define STACKDIR 1           /* 0=grows up; 1=grows down   */
static long newstack[STACKSIZE];

#if (STACKDIR == 1)
#define STACKP (&newstack[STACKSIZE - 1])
#else
#define STACKP newstack
#endif

void CJ_CCPP taskbody(void) {

   if (cjcfjset(&jumpbuffer) == 0)

          /* Switch to new stack and do work              */
          cjcfstkjmp(&jumpbuffer, STACKP,
                    (CJ_VPPROC)dowork);
          /* Never returns to here                        */

   /* Do work using original stack                        */
   dowork(NULL);
   }


void CJ_CCPP dowork(struct cjxjbuf *jbp) {

   /* Do work                                             */

   /* If jump buffer provided, then use long jump to      */
   /* restore the original stack and return               */
   if (jbp != NULL)
          cjcfjlong(jbp, 1);
   }
```

See Also *cjcfstkjmp*

Purpose **Copy a Block of Memory**
Set (Fill) a Block of Memory

These procedures are provided for AMX portability. They are not replacements for C library procedures *memcpy* or *memset*.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZTF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfmcopy(int *sourcep, int *destp,
                        unsigned int size);
void CJ_CCPP cjcfmset(int *mempntr,
                        unsigned int size, int pattern);
```

Description *sourcep* is a pointer to the integer aligned block of memory which is to be copied to the destination.

destp is a pointer to the integer aligned block of memory which is the destination of the block being copied.

mempntr is a pointer to the integer aligned block of memory which is to be filled with *pattern*.

size is the number of integers to be copied or set. The number of bytes copied or set will therefore be *size * sizeof(int)*.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Nothing

Restrictions The source and destination blocks must not overlap unless *destp* is lower in memory than *sourcep*.

ISPs and Timer Procedures should not fill or copy large blocks of memory. Failure to observe this restriction may impose serious performance penalties on your application.

Example

```
#include "CJZZZ.H"

#define BLOCKSIZE 1024
static int srcarray[BLOCKSIZE];
static int dstarray[BLOCKSIZE];


void CJ_CCPP blocksetcopy(int pattern) {

  cjcfmset(srcarray, sizeof(srcarray), pattern);
  cjcfmcopy(srcarray, dstarray, sizeof(srcarray));
  }
```

Purpose **Read or Modify Current Processor Status Register**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_T32U CJ_CCPP cjcfmodcpsr(CJ_T32U mask, CJ_T32U value);
```

Description Use *cjcfmodcpsr* to read/modify the Current Processor Status Register (*CPSR*).

mask is a 32-bit mask defining the bits in the Current Processor Status Register which are to be modified. Set *mask* to *0* to read the Current Processor Status Register without modifying the register.

value is the 32-bit mask defining the values for each of the bits specified by *mask*. If *mask* is *0*, *value* is not used.

Interrupts Procedure *cjcfmodcpsr* does not explicitly affect interrupts. However, the processor interrupt state can be altered by *cjcfmodcpsr* according to the *I* and *F* bit settings in your *mask* and *value* parameters.

Returns *Cjcfmodcpsr* returns the previous state of the Current Processor Status Register (*CPSR*). If *mask* is not *0*, the bits in the *CPSR* register specified by *mask* are updated with the values provided in *value*.

Restrictions Interrupts are not explicitly affected by this procedure. The final interrupt state will depend on the previous value of the *I* and *F* bits in the *CPSR* and the values specified for the *I* and *F* bits by parameters *mask* and *value*.

Restrictions This procedure can also be used to manipulate the Current Processor Status Register (*CPSR*) prior to launching AMX or after exiting from AMX. Therefore, you can call this procedure from your *main()* program before or after your call to *cjkslaunch()*. Of course, your *main()* program must be executing in a privileged mode as required by AMX.

See Also *cjcfdi, cjcfdiprev, cjcfei, cjcfflagrd, cjcfflagwr*

cjcfout8 cjcfout16 cjcfout32

Purpose **Write to an 8, 16 or 32-Bit Output Port**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfout8(void *port, CJ_T32 data);
void CJ_CCPP cjcfout16(void *port, CJ_T32 data);
void CJ_CCPP cjcfout32(void *port, CJ_T32 data);
```

Description *port* is the address of an 8, 16 or 32-bit memory-mapped device output port.

data is the 8, 16 or 32-bit value to be output to the port.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Nothing
Cjcfout8 outputs the least significant 8 bits of *data* to the port.
Cjcfout16 outputs the least significant 16 bits of *data* to the port.
Cjcfout32 outputs the full 32 bits of *data* to the port.

Example

```
#include "CJZZZ.H"

                             /* Console status register   */
#define CONSTAT ((CJ_T8 *)0xFFFA002DL)
                             /* Console data register     */
#define CONDATA ((CJ_T8 *)0xFFFA002FL)


void CJ_CCPP conout(char ch) {

                             /* Wait for ready            */
  while ( (cjcfin8(CONSTAT) & 0x80) == 0 )
         ;

                             /* Write character           */
  cjcfout8(CONDATA, (CJ_T32)ch);
  }
```

See Also *cjcfin8, cjcfin16, cjcfin32*

Purpose **Switch Stacks and Jump to a New Procedure**

This procedure is provided for AMX portability.

Used by ■ Task ☐ ISP ☐ Timer Procedure ☐ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZTF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfstkjmp(void *vp, void *stackp,
                        CJ_VPPROC procp);
```

Description *vp* is a pointer which is passed as a parameter to the new procedure.

stackp is a pointer to a properly aligned block of memory for use as a stack. For the ARM family, the stack must be 32-bit word aligned.

Stackp must point to the top of the memory block since the processor stack builds downward by popular convention.

procp is a pointer to the new procedure which is prototyped as follows:

```
void CJ_CCPP newfunc(void *vp);
```

For portability using different C compilers, cast your procedure pointer as *(CJ_VPPROC)newfunc* in your call to *cjcfstkjmp*.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns There is no return from *cjcfstkjmp*. Use *cjcfjset* and *cjcfjlong* if there is a requirement to return to the original stack.

Restrictions The new procedure referenced by *procp* must never return. The procedure can call *cjtkend* to end the calling task.

Example See the example provided with *cjcfjset* and *cjcfjlong*.

See Also *cjcfjlong, cjcfjset, cjtkend*

Purpose **Convert a String to an Object Name Tag**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZTF.H*.

```
#include "CJZZZ.H"
CJ_TYTAG CJ_CCPP cjcftag(char *tag);
```

Description *tag* is a pointer to a string which is a one to four character name tag.

Interrupts □ Disabled □ Enabled □ Restored

Returns The name tag string is converted to a 32-bit name tag value of type *CJ_TYTAG* which is returned to the caller.

If the name tag string is less than four characters, the returned name tag value is 0 filled. If the name tag string is longer than four characters, the returned name tag value is limited to the first four characters of the string.

Example See any of the *cjXXbuild* examples in which an object name tag string is converted to a name tag value for insertion into the object definition structure.

See Also *cjksfind, cjksgbfind*

Purpose **Fetch a Volatile 8-Bit, 16-Bit, 32-Bit or Pointer Value**

Use these procedures to fetch the content of a volatile variable if the C compiler does not support the C keyword `volatile`. These procedures (or macros) also guarantee that multiple byte fetches will be done in an indivisible fashion.

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.

```
#include "CJZZZ.H"
CJ_T8 CJ_CCPP cjcfvol8(void *varp);
CJ_T16 CJ_CCPP cjcfvol16(void *varp);
CJ_T32 CJ_CCPP cjcfvol32(void *varp);
void * CJ_CCPP cjcfvolpntr(void *pntrp);
```

Description *varp* is a pointer to an 8, 16 or 32-bit variable.

pntrp is a pointer to a pointer variable.

Interrupts □ Disabled □ Enabled □ Restored

Returns *Cjcfvol8* returns an 8-bit signed value from **varp*.
Cjcfvol16 returns a 16-bit signed value from **varp*.
Cjcfvol32 returns a 32-bit signed value from **varp*.
Cjcfvolpntr returns a pointer from **pntrp*.

Example

```
#include "CJZZZ.H"

extern CJ_T8 controlflag;    /* Volatile control flag     */
extern int *valuep;          /* Volatile pointer          */


int * CJ_CCPP readpntr(void) {
  int    *pntr;

                             /* Wait until access allowed */
  while (cjcfvol8(&controlflag) == 0)
         ;

                             /* Wait for valid pointer    */
  while ((pntr = (int *)cjcfvolpntr(&valuep)) == CJ_NULL)
         ;

  controlflag = 0;
  return (pntr);
  }
```

This page left blank intentionally.

Purpose **Fetch Pointer to the AMX Vector Table**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjksivtp(void);
```

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns A pointer to the AMX Vector Table.

See Also *cjksivtrd, cjksivtwr, cjksivtx*

Purpose **Read from the AMX Vector Table**

Used by ■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjksivtrd(int vector, CJ_ISPPROC *oldproc);
```

Description *vector* is the AMX vector number.

AMX vector numbers are assigned by you using *...IRQ* and *...FIQ* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table required to accommodate the total number of devices identified by your *...IRQ* and *...FIQ* directives.

oldproc is a pointer to storage for a copy of the Interrupt Service Procedure pointer retrieved from the specified entry in the AMX Vector Table.

Interrupts □ Disabled □ Enabled □ Restored

Returns Error status is returned.

CJ_EROK	Call successful.

**oldproc* contains the ISP root service procedure pointer retrieved from AMX Vector Table entry number *vector*.

Errors returned:

For all errors, **oldproc* is undefined on return.

CJ_ERRANGE	Invalid AMX vector number.

See Also *cjksivtwr, cjksivtx*

Purpose	**Write to the AMX Vector Table**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjksivtwr(int vector, CJ_ISPPROC newproc);
```

Description

vector is the AMX vector number.
AMX vector numbers are assigned by you using *...IRQ* and *...FIQ* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table required to accommodate the total number of devices identified by your *...IRQ* and *...FIQ* directives.

newproc is a pointer to the ISP root representing the new Interrupt Service Procedure.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns

Error status is returned.

CJ_EROK	Call successful.

Errors returned:

CJ_ERRANGE	Invalid AMX vector number.

See Also *cjksivtrd, cjksivtx*

Purpose **Exchange an Entry in the AMX Vector Table**

Used by ■ Task ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjksivtx(int vector,
                          CJ_ISPPROC newproc,
                          CJ_ISPPROC *oldproc);
```

Description *vector* is the AMX vector number.
AMX vector numbers are assigned by you using *...IRQ* and *...FIQ* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table required to accommodate the total number of devices identified by your *...IRQ* and *...FIQ* directives.

newproc is a pointer to the ISP root representing the new Interrupt Service Procedure.

oldproc is a pointer to storage for the previous ISP root service procedure pointer retrieved from the AMX Vector Table.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful.

**oldproc* contains the previous ISP root service procedure pointer).

Errors returned:
For all errors, **oldproc* is undefined on return.

CJ_ERRANGE	Invalid AMX vector number.

See Also *cjksivtrd, cjksivtwr*

Appendix C. AMX for ARM ROM Option

An AMX system can be configured in two ways. The particular configuration is chosen to best meet your application needs.

Most AMX systems are linked. Your AMX application is linked with your System Configuration Module, your Target Configuration Module and the AMX Library. The resulting load module is then copied to memory for execution either by loading the image into RAM or by committing the image to ROM. Such a ROM contains an image of your application merged with AMX in an inseparable fashion.

The AMX ROM option offers an alternate method of committing AMX to ROM. The ROM option allows the subset of AMX and its managers required by your application to be linked together without any application code to form a separate AMX ROM image. The resulting ROM can be located anywhere in your memory configuration. The penalty paid for ROMing in this fashion is slightly slower access by application code to AMX services.

Selecting AMX ROM Options

To support an AMX ROM system, the following files are provided.

AMX 4-ARM	**AMX 4-Thumb**	
CJ402ROP.LKT	*CJ422ROP.LKT*	AMX ROM Option toolset dependent Link Specification Template
CJ402ROP.CT	*CJ422ROP.CT*	AMX ROM Option Template
CJ402RAC.CT	*CJ422RAC.CT*	AMX ROM Access Template

To use the AMX ROM option, you must edit your Target Parameter File to identify the AMX components which you wish to place in the AMX ROM and to specify where the AMX ROM is to be located. You can use the AMX Configuration Builder to enter these parameters as described in Chapter 4.6.

Creating an AMX ROM

The AMX ROM is created by using the AMX Configuration Generator to produce a ROM Option Module which is then linked with the AMX Library to form an AMX ROM image.

The Configuration Generator combines the information in your Target Parameter File with the ROM Option Template file *CJ402ROP.CT* to produce an assembly language ROM Option Module *CJ402ROP.S*.

You can use the AMX Configuration Builder to generate the ROM Option Module. Use the AMX Configuration Manager to open your Target Parameter File. Make the ROM Option Module selector the active selector. The ROM Option window will become visible allowing you to view your ROM option parameters. To generate the ROM Option Module, select Generate... from the File menu.

If you are unable to use the AMX Configuration Manager or are creating your ROM Option Module from within a make file, you can use the stand alone version of the Configuration Generator. If your Target Parameter File is named *HDWCFG.UP*, the stand alone version of the Configuration Generator utility is invoked as follows:

```
    CJ402CG HDWCFG.UP CJ402ROP.CT CJ402ROP.S
```

The ROM Option Module *CJ402ROP.S* is then assembled in exactly the same manner as your Target Configuration Module *HDWCFG.S* according to the directions in the AMX Tool Guides.

The AMX ROM is linked according to the directions in the AMX Tool Guides.

The resulting AMX ROM image file is then committed to ROM using conventional ROM burning tools. The manner in which this is accomplished will depend completely upon your development environment. In general, the process involves the transfer of the AMX ROM hex file to a PROM programmer.

Note that your toolset may require a filename extension other than *.S* for assembly language files.

Note

Use *CJ402xxx.xxx* files for AMX 4-ARM.
Use *CJ422xxx.xxx* files for AMX 4-Thumb.

Linking for AMX ROM Access

The AMX Configuration Generator is used to produce a ROM Access Module which, when linked with your application, provides access to AMX in the AMX ROM.

The Configuration Generator combines the information in your Target Parameter File with the ROM Access Template file *CJ402RAC.CT* to produce an assembly language ROM Access Module *CJ402RAC.S*.

You can use the AMX Configuration Builder to generate the ROM Access Module. Use the AMX Configuration Manager to open your Target Parameter File. Make the ROM Access Module selector the active selector. The ROM Option window will become visible allowing you to view your ROM option parameters. To generate the ROM Access Module, select Generate... from the File menu.

If you are unable to use the AMX Configuration Manager or are creating your ROM Access Module from within a make file, you can use the stand alone version of the Configuration Generator. If your Target Parameter File is named *HDWCFG.UP*, the stand alone version of the Configuration Generator utility is invoked as follows:

```
CJ402CG HDWCFG.UP CJ402RAC.CT CJ402RAC.S
```

The ROM Access Module *CJ402RAC.S* is then assembled in exactly the same manner as your Target Configuration Module *HDWCFG.S* according to the directions in the AMX Tool Guides.

The AMX ROM Access Module provides access to all of the procedures of AMX and the subset of AMX managers which you included in your AMX ROM. These ROM access procedures make software jumps to the ROM resident procedures.

To create an AMX system which uses your AMX ROM, proceed just as though you were going to include AMX as part of a linked system. Your System Configuration Module must indicate that AMX and its managers are in a separate ROM. To meet this requirement, you may have to use the AMX Configuration Manager to edit your User Parameter File accordingly and regenerate your System Configuration Module. If you do so, do not forget to recompile the System Configuration Module.

Your AMX application is then linked as described in the AMX Tool Guides. However, since AMX and a subset of its managers are in ROM, you must include the AMX ROM Access Module *CJ402RAC.O* in your list of object modules to be linked. By so doing, you will preclude the inclusion of AMX and its managers from the AMX Library *CJ402.A*.

Note that you must still include the AMX Library *CJ402.A* in your link in order to have access to the small subset of AMX procedures which are never installed in your AMX ROM.

Note that your toolset may require filename extensions other than *.O* and *.A* for object and library files.

Once linked, your AMX application can be downloaded into RAM memory in your target hardware configuration. Alternatively, your application can be transferred to ROM using the same techniques that were used to produce the AMX ROM. Regardless of the manner in which your AMX system is loaded into your target hardware, access to the AMX ROM via the ROM Access Module is now possible.

For simplicity, the complexities which you will encounter when trying to commit the C Runtime Library to ROM have been ignored. Refer to your C compiler reference manual for guidance in ROMing C code and data in embedded applications.

Warning!

If your AMX ROM was created without a particular manager, then an AMX fatal exit will occur if your system attempts to access that manager.

Moving the AMX ROM

The AMX ROM is not position independent. Nor is the location of the RAM used by AMX.

To move either, you must edit the AMX ROM option parameters in your Target Parameter File to define the new location of the AMX ROM and its RAM. Reconstruct a new AMX ROM image and burn a new AMX ROM. Then rebuild the AMX ROM Access Module and relink your AMX system with it.

Appendix D. Cache Management

D.1 AMX Cache Services

The ARM™ Reference Architecture includes instruction caches and data caches. However, the cache sizes and control methods are ARM implementation dependent. To accommodate the differences, AMX 4-ARM and AMX 4-Thumb provide a set of cache management services and a mechanism for adapting those services to changing needs.

Manipulating the caches is not trivial and usually requires expertise in assembly language programming. To assist you in the cache setup and use, a cache support module is provided in the AMX Library for each of the supported cache types. These modules include the low level cache control functions needed to manipulate the instruction and/or data caches. These functions are described in Appendix D.2.

The low level cache control functions are NOT dependent on AMX. They can therefore be used to initialize the caches after a power on reset or software reset and to manipulate the caches prior to launching AMX.

AMX also includes a set of high level cache support functions which make use of the low level functions to manipulate caches. These high level functions can be used by applications without an intimate knowledge of the underlying cache architecture.

Cache Enable/Disable

AMX includes a set of high level AMX cache support functions `cjcfhwXcache` to enable or disable the instruction and/or data caches. In the process of enabling or disabling the caches, the selected caches are also flushed and invalidated.

These functions use the low level cache control functions to manipulate the selected caches. These high level functions are located in your Target Configuration Module allowing the instruction and data cache sizes to be automatically adjusted according to the processor or architecture identified in your User Parameter File.

The AMX functions `cjcfhwXcache` are described in Appendix B. These functions can be called either before or after AMX is launched.

Cache Flush and Invalidate

AMX includes a set of high level AMX cache support functions `cjcfhwXflush` to flush and invalidate the instruction and/or data caches without enabling or disabling the caches in the process. Furthermore, these functions do not flush the entire cache. They flush and invalidate only the region specified by you.

The AMX `cjcfhwXflush` functions, also located in the AMX Target Configuration Module, use the low level cache control functions.

> Note
>
> The high level AMX cache support functions do not disable interrupts. It is imperative that these functions be called with interrupts disabled in order to avoid cache conflicts while the caches are being manipulated.

> Warning
>
> If the processor only supports the *random* mode of cache line use, then the low level AMX cache control functions will attempt to invalidate the entire instruction and/or data cache when requested to flush all or part of the cache. If the entire cache cannot be invalidated by issuing a single cache control command, the cache flushing functions will have no effect. In this unusual case, cached memory must be write-through memory for proper operation.

Cache Initialization

When power is first applied to the ARM, the state of the caches is often indeterminate. Most developers will therefore initialize the cache during the power up sequence. The caches are then enabled and the AMX application is launched. Subsequent cache manipulation is rarely required.

The cache support modules include a cache control function *chXXXcache* which can be used to initialize ARM caches of type *XXX*. This function must be called as described in Appendix D.2. If you choose not to use this function, you should examine its implementation to be certain that you have provided an equivalent cache initialization sequence before launching AMX.

The initialization of the caches is often dependent on the particular hardware environment in which the ARM is used. There are often special registers, including the MMU, which must be initialized to provide memory access and define I/O memory regions before cache operations can be performed. In some cases, all such setup must be completed before the low level cache control functions provided with AMX can be used. In other cases, the caches may have to be initialized and disabled before the memory and I/O register setup can be done.

Included with AMX is a board support module for each of the boards on which AMX has been exercised at KADAK. These modules include a board initialization function *chbrdinit* which sets up the board as required for use by KADAK. The function *chbrdinit* includes a call to the low level cache control function *chXXXcache* to initialize and disable the instruction and data caches.

The *main* function in the AMX Sample Program calls the board initialization function *chbrdinit* in the board support module to initialize the board prior to launching AMX. Although *chbrdinit* includes a call to *chXXXcache* to initialize the caches, the call is actually skipped (using a branch instruction to bypass the call) so that caches are not altered unless you so desire. If you want the caches to be initialized and left disabled, edit the board support source file to delete the branch instruction and allow the call to *chXXXcache*. Instructions are provided in the file.

You may choose not to use one of the board support modules provided with AMX but may wish to use the *chXXXcache* function to initialize the caches. If so, you should examine the source code of function *chbrdinit* in the most applicable board support module to see an illustration of the proper use of the *chXXXcache* function.

D.2 Low Level Cache Control Services

Each AMX cache support module contains a low level function `chXXXcache` which can be used to initialize and control the ARM caches. Each module supports a specific cache type. Each cache type is given a name *XXX* identifying a particular ARM processor or architecture which incorporates cache of that type.

The *chXXXcache* function prototype is as follows:

```
void CJ_CCPP chXXXcache(unsigned int command,
                            unsigned long icsize,
                            unsigned long icparam,
                            unsigned long dcsize,
                            unsigned long dcparam);
```

The *chXXXcache* function parameters are used to adapt the operation of the function to the specific needs of a particular ARM processor. In most cases, the parameters simply accommodate different cache sizes for each particular ARM cache type.

The *command* parameter defines the cache operation. It is a bit mask identifying the cache or caches to be affected and the operation to be performed. When used to **enable or disable** the caches, the *command* bit masks are defined as follows:

0x80000000L	Select the instruction cache
0x40000000L	Select the data cache
0x00000001L	*0/1* = disable/enable the selected caches

Parameter *icsize* defines the total size, in bytes, of the instruction cache. Parameter *icparam* is used to identify the instruction cache block (cache line) characteristics.

Parameter *dcsize* defines the total size, in bytes, of the data cache. Parameter *dcparam* is used to identify the data cache block (cache line) characteristics.

When used to **flush** the caches, the *command* bit masks are defined as follows:

0x80000000L	Select the instruction cache
0x40000000L	Select the data cache
0x00000002L	Bit is *1* to flush the selected caches

Parameter *icsize* defines the total size, in bytes, of the memory region to be flushed. Parameter *icparam* is the 32-bit memory address of the region of interest. Parameters *dcsize* and *dcparam* are undefined. If *icsize* is *0*, the entire instruction and/or data cache will be flushed.

Note

The low level AMX cache support functions do not disable interrupts. It is imperative that these functions be called with interrupts disabled in order to avoid cache conflicts while the caches are being manipulated.

The low level AMX cache control functions are located in the following source files.

`CHV4RCAS.C`	ARM v4 System Architecture (CP15) style cache
`CH45_CAS.C`	Samsung S3C4510 cache

The following table summarizes the supported cache types and identifies which type should be used for the various ARM implementations. The parameters listed are the defaults used by AMX unless overridden by you.

Processor/ Architecture	**Function**	***icsize***	***icparam***	***dcsize***	***dcparam***
ARMV4, ARMV4T	*chv4rcache*	*2048*	*0x000C0010*	*1024*	*0x0CCC0010*
ARMV5, ARMV5T (Note 2)	*chv4rcache*	(Note 4)	*0x000C0000*	(Note 4)	*0x0CCC0000*
ARM710, ARM720T, ARM740T	*chv4rcache*	*8192*	*0x00010000*	*8192*	*0* (Note 3)
ARM910T, ARM920T, ARM922T, ARM926EJ, ARM1020E, ARM1022E, ARM1026EJ	*chv4rcache*	(Note 4)	*0x000D0000*	(Note 4)	*0x288C0000*
ARM1136J, ARM1176JZ	*chv4rcache*	(Note 4)	*0xD00D0000*	(Note 4)	*0x2CCC0000*
ARM1156T2	*chv4rcache*	(Note 4)	*0x100D0000*	(Note 4)	*0x2CCC0000*
ARM940T	*chv4rcache*	(Note 4)	*0x00040000*	(Note 4)	*0x20040000*
ARM946E	*chv4rcache*	(Note 4)	*0x000C0000*	(Note 4)	*0x288C0000*
EP7312	*chv4rcache*	*8192*	*0x00010000*	*8192*	*0* (Note 3)
S3C4510	*ch4510cache*	*8192*	*0* (Note 3)	*8192*	*0x00000010*
SA-110	*chv4rcache*	*16384*	*0x00050020*	*16384*	*0x208C0020*
SA-1100	*chv4rcache*	*16384*	*0x00050020*	*8192*	*0x208C0020*
SA-1110	*chv4rcache*	*16384*	*0x00050020*	*8192*	*0x208C0020*
XScale 80200, 80321	*chv4rcache*	(Note 4)	*0x400D0000*	(Note 4)	*0x208C0000*
i.MX1, i.MX21,	*chv4rcache*	(Note 4)	*0x000D0000*	(Note 4)	*0x288C0000*

Note 1: The high level cache service functions in the Target Configuration Module use the cache parameters illustrated above.
The parameters are derived from the processor or architecture identified in your Target Parameter File.

Note 2: You can edit your Target Parameter File (see Appendix D.3) to define alternate cache parameters. You must use this technique to accommodate custom ASICs based on the ARM v4, v4T, v5 or v5T architectures.

Note 3: These processors use a unified instruction and data cache.
The cache line size, if required, is specified by parameter *dcparam*.

Note 4: The cache size is automatically determined by the low level AMX cache control functions.

The cache parameters configured in your Target Configuration Module can be accessed using the private AMX function *cjcfhwpcache* as illustrated in the following example.

```
void CJ_CCPP cjcfhwpcache(void *storagep); /* Function prototype   */

struct {
         unsigned long icsize;
         unsigned long icparam;
         unsigned long dcsize;
         unsigned long dcparam;
         } cacheparam;           /* Storage for cache parameters  */

:
:
cjcfhwpcache(&cacheparam);        /* Fetch cache parameters        */
```

Each *chXXXcache* cache control function operates in a fashion dictated by the cache type. In the sections which follow, each cache type is described.

Type V4R Cache Services

The ARM v4 and v4T instruction and data cache are manipulated using control bits in the Coprocessor 15 (CP15) control registers provided for that purpose. The ARM family uses the MMU registers in CP15 to define the regions of memory to which instruction and data caching apply. These registers must be initialized by you to match your memory system and to meet the needs of your application. The MMU must be initialized before any AMX cache service functions are used.

The cache control function `chv4rcache` is used to clean (write dirty data), flush (invalidate) and enable/disable the instruction and/or data caches as indicated by its `command` parameter. If the instruction cache is selected by parameter `command`, it is disabled and then flushed (invalidated). If the data cache is selected, it is disabled, cleaned if previously enabled and then flushed (invalidated). The selected caches are then enabled if so directed by parameter `command`.

Parameters `icsize` and `dcsize` define the total instruction and data cache sizes respectively.

Parameters `icparam` and `dcparam` are each used to encode two 16-bit parameters. The least significant 16 bits of each parameter defines the number of bytes in each instruction or data cache line. The upper 16 bits provide a bit mask identifying the permissable CP15 cache control operations. The bit masks, defined in the following table, are encoded to specify which of the 18 possible operations specified by CP15 register 7 have been implemented in the processor of interest.

Cache Operation	**Bit#**	`icparam` **mask**	`dcparam` **mask**
Flush ID cache(s)	16	`0x00010000`	
Flush ID single entry	17	`0x00020000`	
Flush I cache	18	`0x00040000`	
Flush I single entry	19	`0x00080000`	
Flush D cache	18		`0x00040000`
Flush D single entry	19		`0x00080000`
Clean ID cache	20	`0x00100000`	
Clean ID single entry	21	`0x00200000`	
Clean D cache	22		`0x00400000`
Clean D single entry	23		`0x00800000`
Clean and Flush ID cache	24	`0x01000000`	
Clean and Flush ID single entry	25	`0x02000000`	
Clean and Flush D cache	26		`0x04000000`
Clean and Flush D single entry	27		`0x08000000`
Flush Prefetch Buffer	28	`0x10000000`	
Drain Write Buffer	29		`0x20000000`
Flush Branch Target Cache	30	`0x40000000`	
Flush Branch Target Entry	31	`0x80000000`	

When you identify your ARM processor by specifying its architecture (**ARM v4 or v4T**), your AMX Target Configuration Module must make an assumption about the type of cache present in your processor. The default is to assume that your v4 or v4T processor has no cache. If your processor has caches, you must customize your use of the AMX cache function *chv4rcache* as described in Appendix D.3.

For example, set *icsize* to *2048* and *dcsize* to *1024* if you have a 2Kb instruction cache and a 1Kb data cache. Set the remaining parameters to define the subset of CP15 cache operations supported by your processor.

An *icparam* parameter of *0x000C0010* would declare that a separate instruction cache exists which can be flushed (invalidated) completely or selectively. Each instruction cache line consists of 16 bytes.

A *dcparam* parameter of *0x0CCC0010* would declare that a separate data cache exists which can be completely or selectively flushed (invalidated) only, cleaned only or both cleaned and flushed. Each data cache line consists of 16 bytes.

The board initialization function *chbrdinit* in the board support module *ADB7T.S* for the ARM Development Board (ARM7TDMI version) illustrates the proper use of the ARM cache control functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* includes a call to *chv4rcache* to initialize the ARM caches.

Note that the call to *chv4rcache*, although present in *chbrdinit*, is actually skipped so that caches are not altered unless you so desire. If you want the caches to be initialized and left disabled, edit source file *ADB7T.S* to allow the call to *chv4rcache*.

The **StrongARM SA-110** processor implements a 16Kb instruction cache and a 16Kb data cache. It follows the ARM System Architecture to control the caches by manipulating the Coprocessor 15 (CP15) control registers provided for that purpose.

When you identify your ARM processor as a StrongARM SA-110, your AMX Target Configuration Module adjusts the *icsize* and *dcsize* parameters to specify a 16Kb instruction cache and a 16Kb data cache.

The *icparam* parameter of *0x00050020* declares that a separate instruction cache exists which can be completely flushed (invalidated) by itself or in conjunction with the data cache. Each instruction cache line consists of 32 bytes.

The default *dcparam* parameter of *0x208C0020* declares that a separate data cache exists which can be completely or selectively flushed (invalidated). It can also be selectively cleaned (dirty bytes written). The processor also provides the ability to drain the write buffer. Each data cache line consists of 32 bytes.

The board initialization function *chbrdinit* in the EBSA-285 Evaluation Board support module *EBSA285.S* illustrates the proper use of the SA-110 cache initialization functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* includes a call to *chv4rcache* to initialize the SA-110 caches.

Note that the call, although present in *chbrdinit*, is actually skipped so that caches are not altered unless you so desire. If you want the caches to be initialized and left disabled, edit source file *EBSA285.S* to allow the call to *chv4rcache*.

The **StrongARM SA-1100** and **SA-1110** processors implement a 16Kb instruction cache and an 8Kb data cache. They follow the ARM System Architecture to control the caches by manipulating the Coprocessor 15 (CP15) control registers provided for that purpose.

When you identify your ARM processor as a StrongARM SA-1100 or SA-1110, your AMX Target Configuration Module adjusts the *`icsize`* and *`dcsize`* parameters to specify a 16Kb instruction cache and an 8Kb data cache.

The *`icparam`* parameter of *`0x00050020`* declares that a separate instruction cache exists which can be completely flushed (invalidated) by itself or in conjunction with the data cache. Each instruction cache line consists of 32 bytes.

The default *`dcparam`* parameter of *`0x208C0020`* declares that a separate data cache exists which can be completely or selectively flushed (invalidated). It can also be selectively cleaned (dirty bytes written). The processor also provides the ability to drain the write buffer. Each data cache line consists of 32 bytes.

The board initialization function *`chbrdinit`* in the Brutus SA-1100 Evaluation Platform support module *`SA1100EP.S`* illustrates the proper use of the SA-1100 cache initialization functions. Function *`chbrdinit`* is called from the AMX Sample Program *`main`* function. *`Chbrdinit`* includes a call to *`chv4rcache`* to initialize the SA-1100 caches.

The board initialization function *`chbrdinit`* in the Intel SA-1110/SA-1111 Board Support module *`SA1111BS.S`* illustrates the proper use of the SA-1110 cache initialization functions. Function *`chbrdinit`* is called from the AMX Sample Program *`main`* function. *`Chbrdinit`* includes a call to *`chv4rcache`* to initialize the SA-1110 caches.

Note that the call, although present in *`chbrdinit`*, is actually skipped so that caches are not altered unless you so desire. If you want the caches to be initialized and left disabled, edit source file *`SA1100EP.S`* or *`SA1111BS.S`* to allow the call to *`chv4rcache`*.

Type S3C4510 Cache Services

The **Samsung S3C4510** processor implements a unified instruction and data cache of 4Kb or 8Kb. The processor provides custom cache control registers.

When you identify your ARM processor as a Samsung S3C4510, your AMX Target Configuration Module adjusts the *`icsize`* and *`dcsize`* parameters to specify an 8Kb instruction cache and an 8Kb data cache.

The *`icparam`* parameter of *`0`* declares that a unified instruction and data cache exists. The *`dcparam`* parameter of *`0x00000010`* declares that each cache line consists of 16 bytes. The cache is a write-through cache. Hence, function *`ch4510cache`* just invalidates the cache (both instruction and data) when requested to flush and invalidate cached memory.

The board initialization function *`chbrdinit`* in the ARM Evaluator-7T Board Support module *`EVAL7T.S`* illustrates the proper use of the S3C4510 cache initialization function. Function *`chbrdinit`* is called from the AMX Sample Program *`main`* function. *`Chbrdinit`* includes a call to *`ch4510cache`* to initialize the S3C4510 cache.

Note that the call, although present in *`chbrdinit`*, is actually skipped so that the cache is not altered unless you so desire. If you want the cache to be initialized and left disabled, edit source file *`EVAL7T.S`* to allow the call to *`ch4510cache`*.

This page left blank intentionally.

D.3 Customizing AMX Cache Services

Unfortunately, although the ARM Architecture does define how caches should be implemented and manipulated, an increasing number of ARM processors are emerging, each with different approaches to cache management.

The cache services provided by AMX accommodate the different cache control mechanisms employed by ARM Ltd. and other silicon vendors in their ARM products. However, new variants will continue to emerge at regular intervals.

At the time AMX 4-ARM and AMX 4-Thumb were first released, all cache management schemes used a subset of the ARM v4 System Architecture (CP15) specification. Most new ARM processors will use this method but may change the cache sizes to meet the needs of particular applications.

To meet these changing requirements, KADAK has provided a cache override facility. To use this feature, edit your Target Parameter File using the AMX Configuration Manager as described in Chapter 4. Make the Target Configuration Module the active selector and go to the Cache property page. Check the box labeled Use custom cache control. In the field labeled Cache function name, enter the name of the low level AMX cache control function *chXXXcache*. Then adjust the cache parameters which will be passed to that function.

For example, to accommodate a 16Kb instruction cache size and an 8Kb data cache size in an ARM v4 or v4T processor which incorporates the cache control mechanism found in the StrongARM, adjust the custom cache parameters as follows.

chv4rcache	Cache function name	(in AMX Library)
16384	Instruction cache size	(16K bytes)
0x00050020	Instruction cache parameter	(32 bytes/line)
8192	Data cache size	(8K bytes)
0x208C0020	Data cache parameter	(32 bytes/line)

The AMX Configuration Manager inserts a *...CACHE* cache override directive into your Target Parameter File. The *...CACHE* directive is described in Appendix A.2. If you are unable to use the AMX Configuration Manager, you will have to use a text editor to edit your Target Parameter File to include this directive. The *...CACHE* directive from the above example is as follows.

```
...CACHE chv4rcache,16384,0x00050020,8192,0x208C0020
```

Note

The cache parameters in the *...CACHE* directive must be provided in a form acceptable to the ARM assembler which you are using. Embedded spaces in expressions are not allowed.

The cache override facility allows the AMX cache control function *chXXXcache* to be replaced by one of your own making with its own set of cache parameters as in the following example.

YOURcache	Cache function name	(your cache procedure)
16384	Instruction cache size	(16K bytes)
16	Instruction cache parameter	(user defined)
8192	Data cache size	(8K bytes)
16	Data cache parameter	(user defined)

The resulting *...CACHE* directive will be as follows.

```
...CACHE YOURcache,16384,16,8192,16
```

Your cache control function must be prototyped just like the AMX *chXXXcache* functions.

```
void CJ_CCPP YOURcache(unsigned int command,
                           unsigned long icsize,
                           unsigned long icparam,
                           unsigned long dcsize,
                           unsigned long dcparam);
```

The high level AMX cache service functions *cjcfhwXcache* will automatically be adjusted to call your cache control function *YOURcache* with the four parameters defined in the *...CACHE* directive. The interpretation of these parameters is entirely up to your procedure *YOURcache*. The *command* parameter will adhere to the standard bit mask values supported by all *chXXXcache* functions (see Appendix D.2).

If you use one of the AMX *chbrdinit* board support functions, you will have to edit it to call your new function *YOURcache*.

Suppressing Low Level Cache Control Services

You can unconditionally suppress the low level AMX cache control functions from your AMX system. To do so, edit your Target Parameter File using the AMX Configuration Manager as described in Chapter 4. Make the Target Configuration Module the active selector and go to the Cache property page. Check the box labeled Suppress all cache support.

If you are unable to use the AMX Configuration Manager, you will have to use a text editor to edit your Target Parameter File to include the following directive in your Target Parameter File.

```
...CACHE NOCACHE
```

Appendix E. Interrupt Management

E.1 Choosing an AMX Interrupt Model

The ARM architecture supports only two interrupt sources: the FIQ fast external interrupt and the IRQ external normal interrupt. AMX supports this simple architecture using any of the four AMX interrupt models defined in Chapter 3.3.

The full interrupt model permits the AMX Interrupt Supervisor to service both the FIQ and IRQ interrupt exceptions. The IRQ and FIQ interrupt models give the AMX Interrupt Supervisor control of only one interrupt exception. The null interrupt model precludes use of either the FIQ or IRQ interrupt by AMX, a rather unusual configuration.

Custom FIQ Interrupt

ARM recommends that the FIQ interrupt be used to service one or more very high speed devices with special service requirements. Devices requiring DMA transfers via software are ideal candidates for such use.

The FIQ interrupt can also be used as a pseudo non-maskable interrupt (NMI) as described in Chapter 3.3.

To use the FIQ interrupt in this fashion, you must select the AMX **IRQ interrupt model**. The FIQ interrupt becomes, in AMX parlance, a nonconforming interrupt with priority above all AMX conforming interrupts.

Of course you could choose the AMX null interrupt model and use the IRQ interrupt as yet another nonconforming interrupt but you would have no access to AMX services from any interrupt handler.

FIQ Interrupt Dedicated to AMX

If any of the devices which generate FIQ interrupts must use AMX services, then the AMX Interrupt Supervisor must have full control over the FIQ interrupt exception. There are two AMX models which meet this requirement.

The AMX **FIQ interrupt model** dedicates just the FIQ interrupt exception to AMX. However, this model precludes the use of the IRQ interrupt for any purpose. The IRQ interrupt MUST go unused. If you used the IRQ interrupt, all conforming AMX FIQ interrupts would be of higher priority than your nonconforming IRQ interrupt, a direct violation of AMX interrupt rules.

The AMX **full interrupt model** dedicates both the FIQ and IRQ interrupt exceptions to AMX. This is your best model to choose to let AMX control the FIQ interrupt without losing the use of the IRQ interrupt. Using this model, all FIQ and IRQ interrupts are serviced by AMX as conforming interrupts.

IRQ Interrupt Dedicated to AMX

If any of the devices which generate IRQ interrupts must use AMX services, then the AMX Interrupt Supervisor must have full control over the IRQ interrupt exception. There are two AMX models which meet this requirement.

The AMX **IRQ interrupt model** dedicates just the IRQ interrupt exception to AMX and leaves the FIQ interrupt free for your custom use. Using this model, all IRQ interrupts are serviced by AMX as conforming interrupts. All FIQ interrupts are considered nonconforming and, as always, are of higher priority than IRQ interrupts.

The AMX **full interrupt model** dedicates both the IRQ and FIQ interrupt exceptions to AMX. Using this model, all IRQ and FIQ interrupts are serviced by AMX as conforming interrupts.

E.2 Interrupt Prioritization and Nesting

There are several ways of viewing interrupt prioritization and nesting on the ARM processor. The simplest is the two levels of priority inherent in the ARM architecture. All devices which generate an FIQ interrupt are automatically of higher priority than the devices which generate IRQ interrupts. The ARM architecture is such that an FIQ interrupt can always preempt an IRQ interrupt because there exists a window of opportunity for an FIQ interrupt which cannot be eliminated.

Another type of prioritization must occur when multiple devices can generate an IRQ (or FIQ) interrupt request. The interrupt source must be identified by software with or without the assistance of an interrupt controller. The order in which the interrupt sources are identified determines the order in which the devices are serviced. This order of service establishes the device priority.

If an interrupt controller which supports software or hardware interrupt masking is used, then a further level of prioritization is possible. In this case, service of one device can be preempted in favour of another device of higher priority, even if both devices generate an IRQ (or FIQ) interrupt request.

AMX supports all of these types of interrupt prioritization and nesting.

Two Priority Levels

In some AMX applications, the two levels of priority inherent in the ARM architecture may be sufficient.

FIQ interrupt handlers execute with both FIQ and IRQ interrupts disabled. Nested interrupts cannot occur.

IRQ interrupt handlers execute with only the IRQ interrupt disabled allowing preemption by an FIQ interrupt. When the AMX full interrupt model is used, nested FIQ interrupts are serviced by AMX as conforming ISPs.

Multiple Software Priority Levels

When multiple devices are multiplexed through the IRQ (or FIQ) interrupt exception, the AMX Interrupt Supervisor calls your IRQ (or FIQ) Interrupt Identification Procedure (IIP) to determine the source of the interrupt. If the IIP must poll the devices to determine the source of the interrupt, the order in which the devices are polled determines the device priority.

If the ARM processor includes an interrupt controller of the type defined in the ARM Reference Peripherals Specification, the order in which the device interrupt request bits in the source status register are examined determines the device priority. The IIPs provided with AMX examine the interrupt status bits from bit 0 (highest priority) to bit 15 (lowest priority).

When multiplexed IRQ devices are simply polled without being masked, nested IRQ interrupts are not truly supported. Since IRQ interrupt handlers execute with the IRQ interrupt disabled, nested IRQ interrupts cannot occur. Each IRQ interrupt will be serviced to completion before another IRQ interrupt will be serviced.

Similarly, when multiplexed FIQ devices are simply polled without being masked, nested FIQ interrupts are not truly supported. Since FIQ interrupt handlers execute with both FIQ and IRQ interrupts disabled, nested FIQ interrupts cannot occur. Each FIQ interrupt will be serviced to completion before another FIQ interrupt will be serviced.

However, FIQ interrupts can preempt IRQ interrupts. Hence, at least one level of interrupt nesting is always possible when both FIQ and IRQ interrupts are used.

Multiple Hardware Priority Levels

An external device such as an Intel 8259 Programmable Interrupt Controller (PIC) can be used to arbitrate interrupt requests and assign interrupt priorities to multiple devices. A single 8259 PIC can support as many as eight devices connected to the FIQ or IRQ interrupt request pins.

If more than eight interrupt sources are required, two 8259 PICs can be used, one connected to each of the FIQ and IRQ pins. However, it may be more advantageous, and just as effective, to connect one master 8259 PIC to the IRQ pin and attach to it one or more slave 8259 PICs. This configuration achieves full interrupt prioritization, allows nested interrupts and still leaves the FIQ interrupt exception free for dedicated, very high priority nonconforming use.

When multiple devices are multiplexed through the IRQ (or FIQ) interrupt exception, the AMX Interrupt Supervisor calls your IRQ (or FIQ) Interrupt Identification Procedure (IIP) to determine the source of the interrupt. Your IIP must interrogate the controller to determine the highest priority device requesting service. Of course, how this is done will be dictated by the manner in which the 8259 PIC is interfaced to the ARM processor.

E.3 Nested Interrupts

Nested Nonconforming FIQ Interrupts

A nested interrupt is always possible when both FIQ and IRQ interrupts are used because an FIQ interrupt can always preempt service of an IRQ interrupt. FIQ interrupts must always be serviced with IRQ interrupts inhibited. If the FIQ interrupt is not serviced by the AMX Interrupt Supervisor, it is considered to be a nonconforming interrupt. If your nonconforming FIQ interrupt handler allows nested FIQ interrupts, it must assume sole responsibility for doing so.

Nested Conforming FIQ or IRQ Interrupts

A conforming AMX Interrupt Service Procedure (ISP) can always be preempted by another conforming ISP. That is, while the AMX Interrupt Supervisor and your ISP are servicing one device interrupt, a second higher priority interrupt request can be acknowledged by the AMX Interrupt Supervisor and serviced by your ISP for that device. Only when service for the second device is complete will service of the first device resume.

When using the **IRQ interrupt model**, a nested conforming interrupt can only occur if more than one device generates IRQ interrupts. Furthermore, interrupt masking must be provided by an interrupt controller or by your IRQ Interrupt Identification Procedure (IIP). Only then can one IRQ interrupt request preempt service of another. Of course, a nonconforming FIQ interrupt can always preempt service of a conforming IRQ interrupt.

When using the **FIQ interrupt model**, a nested conforming interrupt can only occur if more than one device generates FIQ interrupts. Furthermore, interrupt masking must be provided by an interrupt controller or by your FIQ Interrupt Identification Procedure (IIP). Only then can one FIQ interrupt request preempt service of another.

When using the **full interrupt model**, a nested conforming interrupt is always possible because any conforming FIQ interrupt can always preempt service of any conforming IRQ interrupt. Furthermore, when using this model, if more than one device generates FIQ interrupts, a conforming FIQ interrupt can interrupt service of another FIQ interrupt provided that nested FIQ interrupts are supported by your FIQ Interrupt Identification Procedure (IIP). Similarly, if more than one device generates IRQ interrupts, a conforming IRQ interrupt can interrupt service of another IRQ interrupt provided that nested IRQ interrupts are supported by your IRQ IIP.

IRQ Interrupt Handler for Nested Interrupts

An IRQ Interrupt Handler is called by the AMX Interrupt Supervisor with IRQ interrupts disabled. If the AMX Interrupt Supervisor is responsible for FIQ interrupts, FIQ interrupts will be enabled. If not, FIQ interrupts will be enabled or disabled according to your use of FIQ interrupts in your application.

Normally your IRQ Interrupt Handler must execute with IRQ interrupts disabled. However, if nested IRQ interrupts are permitted by your hardware and supported by your IRQ Interrupt Identification Procedure, then your IRQ Interrupt Handler can enable IRQ interrupts. AMX procedures *`cjcfei`* or *`cjcfmodcpsr`* can be used for this purpose.

FIQ Interrupt Handler for Nested Interrupts

An FIQ Interrupt Handler is called by the AMX Interrupt Supervisor with both FIQ and IRQ interrupts disabled.

Normally your FIQ Interrupt Handler must execute with both FIQ and IRQ interrupts disabled. Your FIQ Interrupt Handler must never enable IRQ interrupts. However, if nested FIQ interrupts are permitted by your hardware and supported by your FIQ Interrupt Identification Procedure, then your FIQ Interrupt Handler can enable FIQ interrupts. AMX procedure *`cjcfmodcpsr`* must be used for this purpose. Do not use AMX procedure *`cjcfei`*.

E.4 Interrupt Identification Procedure for Nested Interrupts

For an IRQ interrupt is to preempt service of another IRQ interrupt, your IRQ Interrupt Identification Procedure (IIP) must support nested interrupts. Similarly, for an FIQ interrupt is to preempt service of another FIQ interrupt, your FIQ Interrupt Identification Procedure must support nested interrupts.

In the discussion which follows, the IRQ Interrupt Identification Procedure will be described. The discussion applies equally to the FIQ IIP; differences are specifically identified.

The AMX Configuration Manager inserts a *...PIC* directive into your AMX Target Parameter File to indicate that interrupt nesting is supported by your IRQ and/or FIQ IIP. The IIP calling sequence is described in Chapter 3.2. When nesting is supported, the calling sequence is enhanced to allow your IIP to provide both interrupt identification prior to entry to your Interrupt Handler and interrupt dismissal after service is complete.

Your IRQ Interrupt Identification Procedure is called by the AMX Interrupt Supervisor whenever a new IRQ interrupt is detected. On entry to the IIP, bit 0 of the return address in register *lr* is 0 indicating that a new interrupt is about to be serviced. The IIP must record the priority of the IRQ interrupt currently under service, if any, and identify the new IRQ interrupt source and its priority. The IIP must then inhibit all other IRQ interrupts of priority equal to or less than the new interrupt priority. How this is done depends on the nature of the supporting IRQ interrupt controller and IRQ device interfaces.

When nested IRQ interrupts are supported, your IRQ Interrupt Identification Procedure is also called by the AMX Interrupt Supervisor whenever your device Interrupt Handler finishes service of an IRQ interrupt from that device. On entry to the IIP, bit 0 of the return address in register *lr* is 1 indicating that interrupt service is complete. The IIP must restore the priority of the IRQ interrupt subsystem to reflect conditions prior to the interrupt which has just been serviced. The IIP must inhibit all other IRQ interrupts of priority equal to or less than the restored interrupt priority. Again, how this is done depends on the nature of the supporting IRQ interrupt controller and IRQ device interfaces.

Most IIPs of this kind will require nested storage to retain the interrupt mask history in order to successfully unravel the nested interrupts. Consequently, for each IRQ (FIQ) interrupt, AMX allocates a block of storage, called a history frame, on the AMX Interrupt Stack. Your IIP can use this history frame to save and restore interrupt mask information on entry and exit from each interrupt.

Target Parameter File

For your IRQ (FIQ) IIP to support nesting, you must first edit your Target Parameter File using the AMX Configuration Manager. Make the Target Configuration Module the active selector and go to the Interrupts property page. Check the box for the IRQ (FIQ) option labeled Prioritized/nested by software. In the field labeled IIP storage (bytes), enter the amount of history storage that your IIP requires.

Interrupt Identification Procedure Calling Convention

When nested IRQ (FIQ) interrupts are supported, the following conditions exist upon entry to your IRQ (FIQ) Interrupt Identification Procedure.

The processor is executing in ARM state in supervisor mode.
For IRQ requests, IRQ interrupts are disabled and must remain so.
For IRQ requests, the FIQ interrupt may be enabled or disabled.
For FIQ requests, the IRQ and FIQ interrupts are disabled and must remain so.

The stack pointer in register `sp` references the AMX Interrupt Stack.
A block of `FSIZE` bytes of storage on the stack at the address specified by `sp` is available for the private use of the IIP.
An AMX register frame has been allocated on the stack and can be referenced via register `v1`.

The return address is in register `lr`.
Bit 0 of register `lr` is 0 if a new interrupt is to be identified.
Bit 0 of register `lr` is 1 if interrupt service has been completed.
Registers `a1`, `a2`, `a3`, `a4`, `ip` and `lr` have been saved and are free for use.
The condition flags in the `CPSR` can be altered.
All other registers must be preserved.

Your AMX Target Parameter File defines the size (`FSIZE` bytes) of the history frame required by your Interrupt Identification Procedure to support prioritized interrupt nesting.

The AMX Interrupt Supervisor allocates a history frame of `FSIZE` bytes on its stack prior to calling the IIP. The storage frame is therefore unique for each interrupt. Most IIPs will use the frame to store enough interrupt priority information to permit the state of the interrupt subsystem to be restored after service of an interrupt.

For example, the IIPs provided with AMX for use with the interrupt controller defined in the ARM Reference Peripherals Specification require only 4 bytes of storage. When called upon to identify the source of an interrupt, these IIPs save the current interrupt mask in the dedicated history frame. The device requesting service is then determined and the interrupt mask is adjusted accordingly. When the interrupt has been serviced, the IIP restores the interrupt mask using the value retrieved from the dedicated history frame.

Interrupt Identification Procedure Return Values

The Interrupt Identification Procedure must return the interrupt identification number for the particular device which generated the exception. The interrupt identification number is an index (`0` to `n-1`) into a block of `n` vectors in the AMX Vector Table allocated to the devices which are multiplexed through the ARM IRQ (FIQ) interrupt exception.

The interrupt identification number is returned in register `a1` as follows:

`a1 = i`	Interrupt number (`0` to `n-1`)
`a1 = -1`	Ignore the interrupt
`a1 = -2`	Generate a fatal exception (unidentified interrupt request)

If the IIP returns `a1 = -1` or `-2`, the interrupt mask (or its equivalent) used for prioritizing device interrupts should not be altered. In either case, the AMX Interrupt Supervisor will ignore the device and will not signal an end of interrupt with a second call to the IIP. As a result, any information saved by the IIP in its private frame on the AMX Interrupt Stack will be lost.

Sample Interrupt Identification Procedures which support interrupt nesting are included in the assembly language board support modules provided with AMX.

E.5 Multiplexed ISP for SA-1110/SA-1111 Board

The Intel SA-1110 Development Platform is capable of generating interrupts from a great many sources. The SA-1110 processor on the SA-1110 Development Board includes a 32 bit interrupt controller to control 32 interrupt sources. Of these, one (interrupt 11) is dedicated to mutiplexing 17 interrupts from GPIO pins 11 to 27 inclusive. GPIO pin 25 provides a single interrupt request from three sources on the SA-1111 Development Board. One of these three can be used to multiplex another 64 possible interrupt sources.

Board support module `SA1111BS.S` includes an AMX Interrupt Identification Procedure `chsa1111irq()` which maps all of these interrupt sources into the linear AMX Vector Table as follows.

AMX Vector Number	**Interrupt Source**
0..10	SA-1110 Interrupt Controller bits 0..10
11	GPIO pin 11 (via interrupt contoller bit 11)
12..31	SA-1110 Interrupt Controller bits 12..31
32..47	GPIO pins 12..27 (via interrupt contoller bit 11)
	The following interrupt sources are serviced by the GPIO pin 25 multiplexed ISP in AMX vector 45:
48-111	SA-1111 external interrupts 0..63
112	Ethernet controller on SA-1111 Development Module
113	USAR device on SA-1111 Development Module
114-127	Reserved

Interrupt Identification Procedure `chsa1111irq()` reads the interrupt controller registers and determines which of 32 sources generated the interrupt. Interrupts are ordered in priority from bit 0 (highest) to bit 31 (lowest). If a GPIO pin, multiplexed via interrupt controller bit 11, is the interrupt source, the GPIO registers are examined to determine which of GPIO pins 11 (highest priority) to pin 27 (lowest priority) generated the request. AMX vector number 0 to 47 is returned to the AMX Interrupt Supervisor which then dispatches through the AMX Vector Table to the relevant device ISP Handler.

Support for the SA-1111 interrupts (AMX vectors 48 to 127) is optional. To enable support, find symbol `SA1111_DECODE` in module `SA1111BS.S` and define its value as `1`. Then, using a text editor, edit your AMX Target Parameter File and add the following directive to the file. To omit SA-1111 interrupt support, define `SA1111_DECODE` to have value `0` and omit the directive `...ISPMUX` from your Target Parameter File.

```
...ISPMUX        chsa1111mux,IRQ
```

The AMX Sample Program for the SA-1110/SA-1111 board has SA-1111 interrupt support enabled. Board support module `SA1111BS.S` includes full documentation of the board wiring assumptions upon which it depends. You can examine the Target Parameter File `CJSAMTCF.UP` to see the use of directive `...ISPMUX`. Note that the `...IRQ` directive allocates 128 vectors to the AMX Vector Table.

The `...ISPMUX` directive forces procedure `cj_kpinitmux()` in the Target Configuration Module to call `chsa1111mux()` in module `SA1111BS.S` to initialize AMX vector 45 (GPIO pin 25) to use the multiplexed ISP function `chsa1111isp()` in the same module. This function services each SA-1111 interrupt request by dispatching through one of AMX vectors 48 to 127. Service continues until no pending SA-1111 interrupts remain.