AMX™ PPC32 Target Guide

First Printing: June 1, 1996
Last Printing: November 1, 2007

Copyright © 1996 - 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

Copyright © 1996-2007 by KADAK Products Ltd.
All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of KADAK Products Ltd., Vancouver, B.C., CANADA.

DISCLAIMER

KADAK Products Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability and fitness for any particular purpose. Further, KADAK Products Ltd. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of KADAK Products Ltd. to notify any person of such revision or changes.

TRADEMARKS

AMX in the stylized form and KwikNet are registered trademarks of KADAK Products Ltd.
AMX, AMX/FS, InSight, *KwikLook* and KwikPeg are trademarks of KADAK Products Ltd.
Microsoft, MS-DOS and Windows are registered trademarks of Microsoft Corporation.
PowerPC is a trademark of IBM Corp.
All other trademarked names are the property of their respective owners.

AMX PPC32 TARGET GUIDE

Table of Contents

Page

1. Getting Started with AMX PPC32 1

1.1 Introduction 1
1.2 AMX Files 2
1.3 AMX Nomenclature 4
1.4 AMX PPC32 Target Specifications 5
1.5 Launch Requirements 6
1.6 Using Floating Point 10

2. Program Coding Specifications 11

2.1 Task Trap Handler 11
2.2 Task Scheduling Hooks 12

3. The Processor Interrupt System 13

3.1 Operation 13
3.2 AMX Vector Table and Interrupt Identification Procedure 23
3.3 AMX Interrupt Priority 25
3.4 Conforming ISPs 26
3.5 Nonconforming ISPs 30
3.6 Quick Interrupts 32
3.7 Processor Vector Initialization 33
3.8 MSR in Exception Handlers 35
3.9 Decrementer Interrupt 36

4. Target Configuration Module 39

4.1 The Target Configuration Process 39
4.2 Target Configuration Parameters 43
4.3 Interrupt Service Procedure (ISP) Definitions 60
4.4 Defining a Fast Clock ISP 63
4.5 Null Functions 65
4.6 ROM Option Parameters 66

5. Clock Drivers 69

5.1 Clock Driver Operation 69
5.2 Custom Clock Driver 71
5.3 AMX Clock Drivers 73
5.3.1 Decrementer Clock Driver 73
5.3.2 8254 Clock Driver 75
5.3.3 PPC403 PIT Clock Driver 77
5.3.4 MPC860 PIT Clock Driver 79
5.3.5 MPC8260 PIT Clock Driver 81
5.3.6 MPC823 PIT Clock Driver 83
5.3.7 PPC405 PIT Clock Driver 85
5.3.8 MPC5200 GPT Clock Driver 87
5.3.9 MPC8560 Global Timer Clock Driver 89

AMX PPC32 TARGET GUIDE
Table of Contents (Cont'd)

Appendices

Page

Appendix A. Target Parameter File Specification **A-1**

A.1 Target Parameter File Structure A-1
A.2 Target Parameter File Directives A-3
A.3 Porting the Target Parameter File A-19

Appendix B. AMX PPC32 Service Procedures **B-1**

Appendix C. AMX PPC32 ROM Option **C-1**

Appendix D. Cache Management **D-1**

D.1 AMX Cache Services D-1
D.2 Low Level Cache Control Services D-4
D.3 Customizing AMX Cache Services D-15
D.4 Cache Problems with MMU D-18

AMX PPC32 TARGET GUIDE
Table of Figures

Page

Figure 1.2-1 AMX Include Files 2
Figure 1.2-2 AMX Assembler Source Files 2
Figure 1.2-3 AMX C Source Files 3
Figure 1.4-1 AMX Design Constants 5
Figure 3.1-1 PowerPC Vector Offsets and Masks 14
Figure 3.1-2 AMX Exception Varieties 22
Figure 3.3-1 AMX Interrupt Priorities 25
Figure 4.1-1 Configuration Manager Screen Layout 40
Figure A.1-1 AMX Target Parameter File A-1

1. Getting Started with AMX PPC32

1.1 Introduction

The AMX™ Multitasking Executive is described in the AMX User's Guide. This target guide describes AMX PPC32 which operates in 32-bit mode on the PowerPC™ and all architecturally compatible processors.

Throughout this manual, the term PowerPC refers specifically to the Motorola, IBM and Apple Computer PowerPC family of processors and all processors which meet the PowerPC Architecture specifications. When distinctions are not important, the term PowerPC is used to reference any processor which has the general characteristics of the PowerPC. When distinctions are important, the processors are identified explicitly.

The purpose of this manual is to provide you with the information required to properly configure and implement an AMX PPC32 real-time system. It is assumed that you have read the AMX User's Guide and are familiar with the architecture of the PowerPC processor.

Installation

AMX PPC32 is 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 PPC32 has been tested by KADAK, a separate chapter in the **AMX PPC32 Tool Guide** is provided.

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 PowerPC assembly language.

Figures 1.2-1, 2 and 3 summarize the AMX modules provided with AMX PPC32. 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
CJ382 .H	Generic include file
CJ382APP.H	Custom application definitions
CJ382CC .H	C dependent definitions
CJ382EC .H	AMX error code definitions
CJ382IF .H	C and target interface prototypes
CJ382KC .H	Private AMX constants
CJ382KF .H	AMX service procedure prototypes
CJ382KP .H	Private AMX prototypes
CJ382KS .H	Private AMX structure definitions
CJ382KT .H	Target processor definitions
CJ382KV .H	AMX version specification
CJ382SD .H	AMX application structure definitions
CJ382TF .H	Target dependent prototypes
CJZZZ .H	Copy of generic include file *CJ382.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
CJ382K .DEF	Private AMX assembly language definitions
CJ382KQ .S	Private AMX math procedures
CJ382KR .S	AMX Interrupt Supervisor
CJ382KS .S	AMX Task Scheduler
CJ382MXA.S	Message Exchange Manager constants
CJ382TDC.S	Time/Date Manager constants
CJ382UA .S	Target processor and C support
CJ382UB .S	Target processor and C support

Figure 1.2-2 AMX Assembler Source Files

File Name	Module
CJ382KA .C	Kernel task services
CJ382KB .C *CJ382KBR.C*	General task services
CJ382KC .C *CJ382KCR.C*	Timer Manager
CJ382KD .C *CJ382KDR.C*	Task management services
CJ382KE .C	Task termination services
CJ382KF .C	Suspend/resume task
CJ382KG .C	Time slice services
CJ382KH .C	Task status
CJ382KI .C	Enter and Exit AMX
CJ382KJ .C	General object access
CJ382KK .C	AMX Vector Table access
CJ382KL .C	Private AMX list manipulation
CJ382KM .C	AMX task scheduler hook services
CJ382KX .C	AMX Kernel Task
CJ382CL .C	Circular List Manager
CJ382LM .C	Linked List Manager
CJ382BM .C *CJ382BMR.C*	Buffer Manager
CJ382EM .C *CJ382EMR.C*	Event Manager
CJ382RM .C	Semaphore Manager (resources)
CJ382SM .C *CJ382SMR.C*	Semaphore Manager
CJ382MB .C *CJ382MBR.C*	Mailbox Manager
CJ382MF .C	Flush mailbox and message exchange
CJ382MM .C *CJ382MMR.C*	Memory Manager
CJ382MX .C *CJ382MXR.C*	Message Exchange Manager
CJ382TDA.C	Time/Date Manager
CJ382TDB.C	Time/Date formatter
CJ382UF .C	Launch and leave AMX
CJ382XTA.C	Message exchange task services
CJ382XTB.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*
CJ382xxx.xxx	AMX PPC32 filenames
CJZZZ.H	Generic AMX include file

The generic include file *CJZZZ.H* is a copy of file *CJ382.H* which includes the subset of the AMX PPC32 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 PowerPC processor.

Processor registers are referenced using the software names specified by the PowerPC Architecture rather than the register numbers.

```
r0 - r31, lr, cr, xer, ctr
f0 - f31
```

1.4 AMX PPC32 Target Specifications

AMX PPC32 was initially developed and tested using the Motorola MPC603 processor on the Ultra 603 motherboard platform. However, the AMX PPC32 design criteria fully encompass the PowerPC Architecture specifications.

AMX uses a set of design constants which vary according to the constraints imposed by each target processor. When operating on the PowerPC 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 1024 bytes
CJ_MINIS	Minimum Interrupt Stack is 1024 bytes
CJ_MINTKS	Minimum task storage (including TCB) is 1024 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 PowerPC 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 PowerPC processor. It is assumed that you will have a boot ROM present which configures the PowerPC 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

AMX requires that the processor operate at **supervisor level**. Set *PR* to 0 in the machine state register (*MSR*). This is the default state when the processor is reset.

AMX is delivered ready for use in either big-endian or little-endian mode. To use the processor in **big-endian mode**, set *ILE* and *LE* to 0 in the machine state register (*MSR*). This is the default state when the processor is reset. To operate in little-endian mode, set both *ILE* and *LE* to 1.

All other mode setting bits in the machine state register must be set to meet your specific hardware configuration.

Interrupt State

Interrupts can be enabled or disabled on entry to AMX. Machine state register (*MSR*) bit *EE* disables (*0*) or enables (*1*) 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.

Once AMX has been launched, you must not alter the machine state register directly. This implies that you cannot use any C library processor support procedures to alter the machine state register. Use AMX procedure *cjcfflagrd* or *cjcfmodmsr* to read the machine state register. Then use AMX procedure *cjcfmodmsr* to alter the machine state register.

After AMX has been launched, you can use AMX procedures *cjcfdi* and *cjcfei* to briefly disable and then enable interrupts. To disable and then restore interrupts, use AMX procedures *cjcfdiprev* and *cjcfmodmsr*.

Prior to launching AMX or after AMX has shut down, you can use AMX procedures *cjcfflagrd* and *cjcfflagwr* to read and modify the machine state register *MSR* to disable and/or enable interrupts.

IMPORTANT!

AMX PPC32 does NOT disable interrupts during critical regions of kernel code. Interrupts which occur in such regions are called Quick Interrupts (see Chapter 3.6).

Instruction and Data Caching

The PowerPC Architecture includes instruction caches and data caches.
The cache sizes are PowerPC implementation dependent.

You must be aware that, on processors which implement the PowerPC memory management unit (MMU), successful cache operation will depend on proper setup of the MMU. AMX does not manipulate the memory management unit. For example, if the MMU 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.

You must, prior to launching AMX, ensure that the MMU is properly initialized to condition the instruction and data block address translation registers (*IBATn* and *DBATn*) to meet your hardware memory addressing specifications and caching requirements. You must also condition the machine state register (*MSR*) to enable/disable block address translation for instructions (*MSR* bit *IR*) and/or data (*MSR* bit *DR*).

You will also have to condition the hardware implementation register (*HID0*) to globally enable/disable the instruction cache (*HID0* bit *ICE*) and/or data cache (*HID0* bit *DCE*).

AMX procedures *cjcfhwbatrd* and *cjcfhwbatwr* can be used to read and/or modify the *xBATn* registers. AMX procedure *cjcfmodhid0* can be used to read and/or modify the *HID0* register. An example is provided in the description of AMX procedure *cjcfhwbatrd*.

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 PowerPC processor are described in Appendix D.

Memory Management Unit (MMU)

The PowerPC Architecture includes a Memory Management Unit (MMU) to support a demand-paged virtual memory environment. AMX does not support this use of the PowerPC memory management unit.

If you are using AMX on the Motorola MPC505 or similar processors, this restriction does not apply. These processors do not implement the PowerPC memory management unit and allow direct access to the full address space supported by the particular processor.

AMX does not manipulate the MMU during its normal operation. However, AMX provides service procedures which you can use prior to launching AMX to initialize the MMU Block Address Translation (*BAT*) registers to meet your memory and device addressing requirements.

Your AMX application code and data must reside within the memory address ranges allowed by the particular PowerPC processor which you are using. The PowerPC MMU, 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 for your specific hardware configuration prior to entry to your `main()` program.

> Warning!
>
> Do not enable the memory caches if the MMU has not been initialized to provide proper cached access to memory.

Big or Little Endian

AMX PPC32 is delivered ready for use with the big endian model in which the most significant byte of a word (long) is stored in the lowest byte address. AMX PPC32 will also operate, without modification, on little endian hardware in which the least significant byte of a word (long) is stored in the lowest byte address. However, to use AMX on little endian hardware, you must first rebuild the AMX Library for little endian operation as described in Appendix D of the AMX User's Guide.

To use the big endian model, set *ILE* and *LE* to 0 in the machine state register (*MSR*). This is the default state when the processor is reset. To use the little endian model, set both *ILE* and *LE* to 1. Refer to the topic "Operating Mode" presented earlier in this chapter.

1.6 Using Floating Point

To improve performance, AMX PPC32 does not preserve the 32 floating point registers as part of a task's context. However, there are techniques which you can use effectively when floating point support is required.

The simplest solution, if feasible, is to restrict the use of floating point operations to a single task. If only one task uses floating point, there can be no conflict.

Alternatively, if you can guarantee, by mutual exclusion with an AMX resource semaphore or by design, that only one task at a time can perform floating point operations, there will be no need to preserve floating point registers for each task.

However, if multiple tasks must perform floating point operations concurrently, then the floating point registers must be preserved for these tasks. To do so, you must allocate a static *cjxfpregs* structure for each task which uses floating point. For convenience, you can save the pointer to the structure for a particular task in the private user area provided in the Task Control Block for that task.

You must install a set of application Task Scheduling Hooks to save and restore the floating point registers whenever tasks which use them are suspended or resumed. These procedures are described in Chapter 14.3 of the AMX User's Guide and must be written as illustrated in Chapter 2.2 of this guide.

Since the floating point registers are PowerPC implementation dependent, two target processor dependent procedures, *cjcffpregst* and *cjcffpregld*, are provided by AMX in your AMX Target Configuration Module. These procedures, which can be called from C or assembly language, can be used by your Task Scheduling Hooks, as described in Chapter 2.2, to preserve the floating point registers across task context switches.

2. Program Coding Specifications

2.1 Task Trap Handler

Unlike other processors, the PowerPC processor does NOT provide exception traps for faults such as arithmetic overflow, integer division by zero or array bounds violations.

Consequently, AMX PPC32 does NOT support task traps.

> Note
>
> 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:

Interrupts are enabled but only quick interrupts are allowed.
The stack pointer in register `r1` references the task's stack.
The Task Control Block address is in register `r4`.
The return address is in register `lr`.
Registers `r3` to `r9`, `lr` and `cr` are free for use.
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 header file `CJ382K.DEF` in your assembly language module, you can reference the private region within the TCB reserved for your use as `XTCBUSER(r4)`.

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

If your tasks do floating point operations, you may have to preserve the floating point registers across task context switches. Your task suspend and resume hooks can do so by calling procedures `cjcffpregst` and `cjcffpregld` respectively as illustrated in the following example. Note that it is assumed that you have allocated a `cjxfpregs` structure for storage of the floating point registers and installed a pointer to that storage into the task block at `XTCBUSER(r4)`. It is further assumed that the pointer is present in the task block if, and only if, the task uses floating point.

```
lwz      r3,XTCBUSER(r4)          # r3 = A(cjxfpregs structure)
cmpwi    r3,0
beqlr                             # If no floating point, return
                                  # Save (restore) floating point
                                  # registers and return
b        cjcffpregst (or cjcffpregld)
```

3. The Processor Interrupt System

3.1 Operation

The PowerPC Architecture 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 at a location in the processor Exception Vector Table. The exact location is determined by adding the unique vector offset for the particular exception to the base address of the Exception Vector Table. The vector offsets (see Figure 3.1-1) are defined in AMX header file *CJ382KT.H*.

The base address of the PowerPC Exception Vector Table is implementation dependent. The PowerPC Architecture specifies that the base address is determined by the state of the *IP* bit in the machine state register (*MSR*). The base address is *0x00000000* (*IP = 0*) or *0xFFF00000* (*IP = 1*). However, some PowerPC implementations, such as the IBM PPC403, provide a special Exception Vector Prefix Register which can be used to relocate the base address. Others, such as the Motorola MPC602, provide an Interrupt Base Register which can relocate the base address of most (but not all) exception vectors.

Processors based on the PowerPC Book E specification, such as the Freescale MPC85xx family or the IBM PPC440 family, provide a special Interrupt Vector Prefix Register which can be used to relocate the base address. Additionally, these processors provide a set of Interrupt Vector Offset Registers (*IVORnn*) which can be used to change the vector offset of each exception vector. To be compatible with AMX, these registers **must be programmed** to match standard PowerPC vector offsets identified in Figure 3.1-1. When AMX is launched, the *IVORnn* registers associated with the exceptions AMX will handle, as specified in the AMX Target Parameter File, will be automatically configured with the offsets listed in Figure 3.1-1.

The fragment of code which resides in an entry in the processor's Exception Vector Table is called an **exception handler**. In most cases, the exception handler simply saves a few registers and dispatches to an application procedure which actually services the exception. Within the AMX environment, the particular procedures which service internal or external device interrupt requests are called **Interrupt Service Procedures**. All other procedures are referred to as **exception service procedures**.

In most PowerPC configurations, the Exception Vector Table is located in RAM at address *0x00000000*. Consequently, the exception handler code for each exception must be installed into the table at program load time or at program run time. Fortunately, AMX will automatically create and install exception handlers for you, eliminating any need to program in PowerPC assembly language. You simply define each exception for which a handler is required and, using simple directives in your AMX Target Parameter File (see Chapter 4), instruct AMX as to how that exception is to be serviced.

Figure 3.1-1 PowerPC Vector Offsets and Masks

Vector Name	Vector Offset	Vector Mask	Exception	
PowerPC Architecture				
CJ_PRVNRES	*0x0100*	*0x00000002*	System reset	
CJ_PRVNMC	*0x0200*	*0x00000004*	Machine check	
CJ_PRVNDA	*0x0300*	*0x00000008*	Data access	(core defined)
CJ_PRVNIA	*0x0400*	*0x00000010*	Instruction access	(core defined)
CJ_PRVNEI	*0x0500*	*0x00000020*	External interrupt	
CJ_PRVNAL	*0x0600*	*0x00000040*	Alignment	
CJ_PRVNPG	*0x0700*	*0x00000080*	Program	
CJ_PRVNFPU	*0x0800*	*0x00000100*	Floating point unavailable	
CJ_PRVNDEC	*0x0900*	*0x00000200*	Decrementer	
	0x0A00	*0x00000400*	reserved (Note 1)	
	0x0B00	*0x00000800*	reserved (Note 2)	
CJ_PRVNSC	*0x0C00*	*0x00001000*	System call	
CJ_PRVNTR	*0x0D00*	*0x00002000*	Trace	
CJ_PRVNFPA	*0x0E00*	*0x00004000*	Floating point assist	
	0x0F00	*0x00008000*	reserved	
0x1000 to	*0x2FFF*		reserved	
PowerPC 601		*0x000013FE*	Common PowerPC exceptions	
CJ_PRVNIOER	*0x0A00*	*0x00000400*	I/O controller interface error	
CJ_PRVNRMTR	*0x2000*	Note 3	Run mode/trace	
PowerPC 602, 603, 5200, 8240, 8260, 8280, 8349				
		0x000033FE	Common PowerPC exceptions	
CJ_PRVNCRIT	*0x0A00*	*0x00000400*	Critical interrupt	(5200, 8280, 8349 only)
CJ_PRVNTMIN	*0x1000*	*0x00010000*	Instruction translation miss	
CJ_PRVNTMDL	*0x1100*	*0x00020000*	Data load translation miss	
CJ_PRVNTMDS	*0x1200*	*0x00040000*	Data store translation miss	
CJ_PRVNIABR	*0x1300*	*0x00080000*	Instruction address breakpoint	
CJ_PRVNSMI	*0x1400*	*0x00100000*	System management interrupt	
CJ_PRVNWD602	*0x1500*	*0x00200000*	Watchdog Timer	(602 only)
CJ_PRVNEM602	*0x1600*	*0x00400000*	Emulation trap	(602 only)
PowerPC 604, 740, 74xx, 750		*0x000033FE*	Common PowerPC exceptions	
CJ_PRVNPM	*0x0F00*	*0x00008000*	Performance monitoring interrupt	
CJ_PRVNAVU	*0x0F20*	Notes 3, 5	AltiVec unavailable	(74xx only)
CJ_PRVNTMIN	*0x1000*	*0x00010000*	Instruction translation miss	(74xx only)
CJ_PRVNTMDL	*0x1100*	*0x00020000*	Data load translation miss	(74xx only)
CJ_PRVNTMDS	*0x1200*	*0x00040000*	Data store translation miss	(74xx only)
CJ_PRVNIABR	*0x1300*	*0x00080000*	Instruction address breakpoint	
CJ_PRVNSMI	*0x1400*	*0x00100000*	System management interrupt	
CJ_PRVNAVA	*0x1600*	*0x00400000*	AltiVec assist	(74xx only)
CJ_PRVNTHMI	*0x1700*	*0x00800000*	Thermal management interrupt	(not on 604)

(continued on next page)

Vector Name	Vector Offset	Vector Mask	Exception	

(continued from previous page)

PowerPC 401, 403, 405		*0x000010FC*	Common PowerPC exceptions	
CJ_PRVNCRIP	*0x0100*	*0x00000002*	Critical interrupt pin	
CJ_PRVNPV	*0x0300*	*0x00000008*	Protection violation	(PPC403GA only)
CJ_PRVNAPUU	*0x0F20*	Notes 3, 5	APU unavailable	(PPC405B3 only)
CJ_PRVNPIT	*0x1000*	Notes 1, 3	Programmable Interval Timer	
CJ_PRVNFIT	*0x1010*	Notes 2, 3	Fixed Interval Timer	
CJ_PRVNWD403	*0x1020*	Note 3	Watchdog Timer	
CJ_PRVNMD403	*0x1100*	*0x00020000*	Data TLB miss	(core defined)
CJ_PRVNMI403	*0x1200*	*0x00040000*	Instruction TLB miss	(core defined)
CJ_PRVNDBG	*0x2000*	Note 3	Debug exception	

PowerPC 505, 509, 555		*0x000073E6*	Common PowerPC exceptions	
CJ_PRVNEM505	*0x1000*	*0x00010000*	Software emulation	
0x1100 to	*0x1BFF*		reserved	
CJ_PRVNIPE	*0x1300*	*0x00080000*	Instruction protection error	(555 only)
CJ_PRVNDPE	*0x1400*	*0x00100000*	Data protection error	(555 only)
CJ_PRVNDBP	*0x1C00*	*0x10000000*	Data breakpoint	
CJ_PRVNIBP	*0x1D00*	*0x20000000*	Instruction breakpoint	
CJ_PRVNMBP	*0x1E00*	*0x40000000*	Maskable external breakpoint	
CJ_PRVNNMBP	*0x1F00*	*0x80000000*	Non-maskable external breakpoint	

PowerPC 8xx (Note 4)		*0x000073FE*	Common PowerPC exceptions
CJ_PRVNEM860	*0x1000*	*0x00010000*	Software emulation
CJ_PRVNTMI	*0x1100*	*0x00020000*	Instruction TLB miss
CJ_PRVNTMD	*0x1200*	*0x00040000*	Data TLB miss
CJ_PRVNTEIN	*0x1300*	*0x00080000*	Instruction TLB error
CJ_PRVNTED	*0x1400*	*0x00100000*	Data TLB error
0x1500 to	*0x1BFF*		reserved
CJ_PRVNDBP	*0x1C00*	*0x10000000*	Data breakpoint
CJ_PRVNIBP	*0x1D00*	*0x20000000*	Instruction breakpoint
CJ_PRVNPBP	*0x1E00*	*0x40000000*	Peripheral breakpoint
CJ_PRVNNMDP	*0x1F00*	*0x80000000*	Non-maskable development port

(continued on next page)

Vector Name	Vector Offset	Vector Mask	Exception	
(continued from previous page)				
PowerPC 555x, 85xx, 440		*0x000013FC*	Common PowerPC exceptions	
CJ_PRVNCRIP	*0x0100*	*0x00000002*	Critical interrupt	(555x, 440 only)
CJ_PRVNPM	*0x0F00*	*0x00008000*	Performance monitoring	(85xx only)
CJ_PRVNAPUU	*0x0F20*	Notes 3, 5	APU unavailable	
	0x1000	Notes 1, 3	reserved	
CJ_PRVNFIT	*0x1010*	Notes 2, 3	Fixed Interval Timer	
CJ_PRVNWD403	*0x1020*	Note 3	Watchdog Timer	
CJ_PRVNMD403	*0x1100*	*0x00020000*	Data TLB error	
CJ_PRVNMI403	*0x1200*	*0x00040000*	Instruction TLB error	
CJ_PRVNSPEU	*0x1600*	*0x00400000*	SPE unavailable	(555x, 85xx only)
CJ_PRVNSPED	*0x1700*	*0x00800000*	SPE floating-point data	(555x, 85xx only)
CJ_PRVNSPER	*0x1800*	*0x01000000*	SPE floating-point round	(555x, 85xx only)
CJ_PRVNDBG	*0x2000*	Note 3	Debug exception	

Note 1: Short vector *0x1000* is redirected to unused vector *0x0A00*.
Note 2: Short vector *0x1010* is redirected to unused vector *0x0B00*.
Note 3: No vector mask is defined. A special form of the *...EVFATAL* directive (see Appendix A.2) is used to force AMX to treat these vectors as fatal.
Note 4: Includes the MPC801, 821, 823, 850, 855 and 860.
Note 5: Vector *0x0F00* is redirected to unused vector *0x1F00*.

Figure 3.1-1 PowerPC Vector Offsets and Masks (continued)

Default Exception Service Procedures

AMX maintains entries in the Exception Vector Table for all of the processor exceptions for which AMX is given responsibility. Your Target Parameter File (see Chapter 4) specifies which of the possible exceptions you wish AMX to service. AMX can be directed to treat each exception vector in one of the following ways:

Fatal exception
Custom user exception
Multiplexed or dedicated device interrupt exception

Specific exceptions can be declared as fatal. Typically these are exceptions defining drastic error conditions (such as memory violations) from which recovery is not possible. During testing, you should only declare fatal those exceptions which are not serviced by your debugger.

Applications requiring direct control over a specific exception can use the AMX Configuration Builder to install a directive in the AMX Target Parameter File (see Chapter 4) to instruct AMX to create and install an exception handler which will dispatch directly to a custom user exception service procedure.

Exceptions caused by device interrupts are revectored by AMX through the AMX Vector Table to device specific handlers as described later in this chapter. Directives in the AMX Target Parameter File (see Chapter 4) identify the exceptions which are to be treated in this fashion.

Finally, AMX will ignore any exception which is not explicitly declared in your AMX Target Parameter File as under AMX control.

Fatal Exception Service Procedures

The AMX Configuration Builder allows you to identify the exceptions in the PowerPC Exception Vector Table which you wish AMX to service and treat as fatal.

A 32-bit vector mask is used by AMX to identify each of the first 32 exception vectors in the Exception Vector Table. The vector mask bits are allocated as defined in Figure 3.1-1. The vector mask is used in the *...EVFATAL* directive which the Configuration Builder inserts in your Target Parameter File to identify the exceptions which you wish AMX to treat as fatal.

Vector masks are not defined for exception vectors at vector offsets *0x2000* and above or for any of the short exception vectors such as *0x1000*, *0x1010* and *0x1020* for the IBM PPC403. Even though vector masks are not defined for these vectors, you can still force AMX to treat them as fatal. The Configuration Builder uses the special forms of the *...EVFATAL* directive described in Appendix A.2.

When a fatal exception is declared, AMX calls its Fatal Exception Procedure *cjksfatalexh* in AMX module *CJ382UF.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 default Fatal Exception Procedure provided with AMX simply returns to the AMX exception handler with the AMX fatal exit code *CJ_FETRAP* indicating that a fatal exception has occurred. AMX then passes the fatal exit code on to the Fatal Exit Procedure *cjksfatal* (also in module *CJ382UF.C*) signifying that a fatal exception has been detected and serviced but deemed unrecoverable.

If the Fatal Exception Procedure *cjksfatalexh* returns an error code of *CJ_EROK* to AMX indicating that the cause of the exception has been remedied, AMX will restore the machine state and resume execution.

Fatal Exception Procedure

The Fatal Exception Procedure, located in module *CJ382UF.C*, is written in C as follows. Upon entry, the content of the machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8. Interrupts are disabled (*EE = 0*).

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

CJ_ERRST CJ_CCPP cjksfatalexh(
struct cjxregs *regp,                       /* A(Register structure)    */
int          vofs)                          /* Vector offset            */
{
   :
   Process the error
   :
   if (the exception cause has been acknowledged and repaired)
            return (CJ_EROK);

   return (CJ_FETRAP);
   }
```

The state of each 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 header file *CJ382KT.H*.

The program address at the time of the fault, as determined by the content of the *srr0* register immediately after the exception, is provided in the register array at *regp->xreg_srr0*.

The machine state register at the time of the exception, as determined by the content of the *srr1* register immediately after the exception, is provided in the register array at *regp->xreg_srr1*.

The register values in structure *cjxregs* can be examined and, in rare circumstances and with extreme care, can be modified. If the Fatal Exception Procedure returns error code *CJ_EROK* to AMX, execution will resume with registers set according to the values in the structure referenced by *regp*.

Execution resumes at the address specified by *regp->xreg_srr0* with the *MSR* register set to the value from *regp->xreg_srr1*.

The Fatal Exception Procedure executes in the machine context in effect at the time the exception occurred. Hence, the procedure may be executed while in an application task, while servicing a device interrupt or while in the AMX kernel. Consequently, the procedure is NOT able to use any AMX services except those (such as list manipulation or processor support) which do not depend on the integrity of private AMX data.

The parameter *vofs* identifies the exception. *Vofs* is an integer of the form *0xvvvnnnnn* where *0xvvv00000* is an AMX **exception variety identifier** and *0x000nnnnn* is a variety dependent parameter. The AMX exception variety identifiers, defined in AMX header file *CJ382KT.H*, are summarized in Figure 3.1-2 later in this chapter.

User Exception Service Procedures

A custom user exception service procedure can be written in C as follows. Upon entry, the content of the machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8. Interrupts are disabled (*EE = 0*).

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

void CJ_CCPP userexcept(
struct cjxregs *regp,                       /* A(Register structure)    */
int         vofs)                           /* Vector offset            */
{
   :
   Process the exception
   :
   }
```

If the exception service procedure defined in your Target Parameter File (see Chapter 4) requires a full register save, the state of each 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 header file *CJ382KT.H*.

If a full register save is not required, an alternate, faster form of user exception service procedure can be defined. In this case, only the subset of registers needed to service the exception and call a C procedure are preserved in register structure *cjxregs*. Note that the saved register subset may vary for C compilers from different vendors.

The program address at the time of the fault, as determined by the content of the *srr0* register immediately after the exception, is provided in the register array at *regp->xreg_srr0*.

The machine state register at the time of the exception, as determined by the content of the *srr1* register immediately after the exception, is provided in the register array at *regp->xreg_srr1*.

Upon return, execution resumes at the address specified by *regp->xreg_srr0* with the *MSR* register set to the value from *regp->xreg_srr1*.

The user exception service procedure executes in the machine context in effect at the time the exception occurred. Hence, the procedure may be executed while in an application task, while servicing a device interrupt or while in the AMX kernel. Consequently, the procedure is NOT able to use any AMX services except those (such as list manipulation or processor support) which do not depend on the integrity of private AMX data.

The parameter *vofs* identifies the exception. *Vofs* is an integer of the form *0xvvvnnnnn* where *0xvvv00000* is the AMX **exception variety identifier** *CJ_PRXVUSR* and *0x000nnnnn* is a user defined parameter, often just the vector offset value. The AMX exception variety identifiers, defined in AMX header file *CJ382KT.H*, are summarized in Figure 3.1-2 later in this chapter.

For even faster response, a custom user exception service procedure written in assembly language can be defined. Upon entry to the procedure, the following conditions exist.

The machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8.
Interrupts are disabled (*EE = 0*) and must remain so.
The stack pointer in register *r1* references the stack in effect at the time the exception occurred.
A register frame has been allocated on the stack and can be referenced via register *r1*.
Registers *r28*, *r31*, *lr* and *cr* have been saved and are free for use.
The return address is in register *lr*.
Register *r31* contains the *vofs* parameter *0xvvvnnnnn* previously described.
All other registers must be preserved.

If you include AMX header file *CJ382K.DEF* in your assembly language module, you can use 28 words (112 bytes) in the register frame at *XREG_R(r1)* for temporary storage.

Device Interrupt Service Procedures

A subset of the exception vectors are reserved for the control of interrupt generating devices external to or embedded in the processor. These vectors include, but are not limited to, the following.

Vector Offset	Exception
0x0500	External interrupt
0x0900	Decrementer interrupt
0x1400	System management interrupt
0x1500	Watchdog timer

The external and internal interrupt facility is controlled by setting the *EE* bit in the machine state register (*MSR*) to *0* to disable interrupts or to *1* to enable interrupts.

When an interrupt is acknowledged, the processor moves the address of the current instruction into register *srr0* and copies the machine state register (*MSR*) into register *srr1*. The interrupt enable bit (*EE*) in the *MSR* is then reset to *0* thereby disabling all external interrupts. The processor then begins execution of the exception handler in the Exception Vector Table at the vector offset determined by the particular interrupt source.

When multiple devices must generate interrupts through a single PowerPC exception vector, external hardware (such as an Intel 8259 Interrupt Controller) must be used to arbitrate the requests. Some PowerPC implementations include an embedded interrupt controller to arbitrate internal and external interrupt requests. In either case, the interrupt exception handler must distinguish the particular interrupt source, service the device and remove the interrupt request.

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. AMX will automatically create and install an interrupt dispatching exception handler into the Exception Vector Table for you, eliminating any need to program in PowerPC assembly language. You simply define, through your AMX Target Parameter File (see Chapter 4), each interrupt exception for which such a handler is required and instruct AMX as to how the devices attached to that exception are to be serviced.

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 even though an AMX exception handler may be used to dispatch to the device service routine.

AMX Exception Varieties

Fatal Exceptions

Fatal exceptions are those which you have explicitly requested AMX to treat as fatal. For example, if you do not have any source of external interrupts, you might let AMX service the *0x0500* external interrupt exception as a fatal exception. Then, if such an exception ever occurs, AMX will service the exception and pass it on to the Fatal Exception Procedure.

Fatal Pseudo Exceptions

As will be explained in Chapter 3.2, AMX allows you to vector the service of external and internal device interrupts through an AMX Vector Table. In response to unexplained events occurring during its service of device interrupts, AMX will declare a fatal exception and call its Fatal Exception Procedure. These fatal exceptions are called pseudo exceptions because they result from actions initiated by AMX.

If the device which generated an interrupt exception cannot be identified, AMX generates a pseudo exception of variety *CJ_PRXVUIR* signifying an unidentified interrupt request. If the device which generated an interrupt exception can be identified but an application device handler has not been provided for it, AMX generates a pseudo exception of variety *CJ_PRXVUIV* signifying that an interrupt occurred for an uninitialized interrupt vector.

User Exceptions

User exceptions are those which you have requested AMX to direct to your custom user exception service procedure. For example, if you wish to service the system call exception yourself, you can direct AMX to treat the *0x0C00* system call exception as a user exception. Then, whenever such an exception occurs, AMX will service the exception and pass it on to your user exception service procedure.

Exception Variety	**Variety Parameter** (*nnnnn*)	**Description**
CJ_PRXVFEX	vector offset	Fatal exception serviced via AMX
CJ_PRXVUIR	vector offset	Unidentified interrupt request
CJ_PRXVUIV	vector number	Uninitialized interrupt vector
CJ_PRXVUSR	user defined	User exception
CJ_PRXVMASK		Mask used to isolate exception variety Use *~CJ_PRXVMASK* to isolate the parameter

Figure 3.1-2 AMX Exception Varieties

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 decrementer exception (exception vector offset *0x0900*). Then a block of 16 vectors starting at AMX vector number 8 might be allocated for up to 16 devices attached to the external interrupt exception (exception vector offset *0x0500*). In this case, the AMX Vector Table would include 24 entries, seven of which are unallocated (vectors 0 and 2 to 7).

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 exception variety *CJ_PRXVUIV* (see Figure 3.1-2) to identify the vector through which the interrupt for the unsupported device was to be serviced.

Interrupt Identification Procedure (IIP)

When multiple devices generate interrupts through a single PowerPC exception vector, you must provide an **Interrupt Identification Procedure** which the AMX exception handler 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 PowerPC assembly language.

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

> The machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8.
> Interrupts are disabled and must remain so.
> The stack pointer in register *r1* references the stack in effect at the time the interrupt exception occurred.
> A register frame has been allocated on the stack and can be referenced via register *r1*.
> Registers *r28*, *r31*, *lr* and *cr* have been saved and are free for use.
> The return address is in register *lr*.
> 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 single PowerPC exception vector.

The interrupt number is returned in register *r31* as follows:

r31 = i	Interrupt number (*0* to *n-1*)
r31 = -1	Ignore the interrupt
r31 = -2	Generate a fatal exception of variety *CJ_PRXVUIR* (unidentified interrupt request; see Figure 3.1-2)

If you include AMX header file *CJ382K.DEF* in your assembly language module, you can use 28 words (112 bytes) in the register frame at *XREG_R(r1)* 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 *r31 = -2*, AMX generates a fatal exception of variety *CJ_PRXVUIR* to identify the interrupt exception through which the spurious interrupt occurred.

> Warning
>
> Your Interrupt Identification Procedure must not modify the *XER* register without saving and restoring it. Hence, avoid using instructions which modify the carry flag.

3.3 AMX Interrupt Priority

The PowerPC family of processors provides limited interrupt priority ordering. Interrupt exceptions are serviced in the following order of priority.

Vector Offset	**Priority**	**Exception**
`0x1400`	highest	System management interrupt (PowerPC 602, 603, 604)
`0x0500`		External interrupt
`0x0900`		Decrementer interrupt
`0x1500`	lowest	Watchdog timer (PowerPC 602)

Figure 3.3-1 AMX Interrupt Priorities

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

Priority ordering can also be achieved without hardware arbitration. Your Interrupt Identification Procedure for a particular exception establishes the priority of the devices which generate that exception. In general, the priority is determined by the order in which the IIP polls the devices to determine the interrupt source.

Finally, 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.

3.4 Conforming ISPs

A conforming ISP consists of three parts: an ISP root, an ISP stem and an ISP 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 ISP stem is a minimal handler responsible only for dismissing the device interrupt as rapidly as possible. The ISP Handler continues the device service initiated by the ISP stem.

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.

When the interrupt occurs, the AMX exception handler calls your Interrupt Identification Procedure to determine the source of the interrupt. Using the interrupt identification number provided by the IIP, the AMX exception handler indexes into the AMX Vector Table to find the ISP root associated with the particular device.

The ISP root is the actual Interrupt Service Procedure which is executed by the AMX exception handler. The ISP root calls the ISP stem to dismiss the interrupt request and service the hardware device. The ISP stem executes with all interrupts disabled.

The manner in which the ISP stem returns to the ISP root determines whether the ISP Handler has to be invoked. If the ISP Handler can be bypassed, the ISP root simply resumes execution at the original point of interruption.

If the ISP Handler must be executed, the ISP root queues a request to the AMX Interrupt Supervisor to execute the ISP Handler. If it has not already done so, the Interrupt Supervisor switches to the AMX Interrupt Stack.

The AMX Interrupt Supervisor then calls the ISP Handler to complete the interrupt servicing sequence. Upon return from the ISP Handler, 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 ISP Handler.

The ISP stem and ISP Handler can be written 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).

The ISP stem can be written in assembly language or C. No AMX task synchronization or communication services can be used.

Upon entry to your assembly language ISP stem, the following conditions exist:

The machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8.
Interrupts are disabled and must remain so.
The stack pointer in register *r1* references the stack in effect at the time the interrupt exception occurred.
A register frame has been allocated on the stack and can be referenced via register *r1*.
Registers *r28*, *r31*, *lr* and *cr* have been saved and are free for use.
The return address is in register *lr*.
r31 contains the ISP parameter, if any.
All other registers must be preserved.

Upon return, *r31* contains a continuation directive.

If you include AMX header file *CJ382K.DEF* in your assembly language module, you can use 28 words (112 bytes) in the register frame at *XREG_R(r1)* for temporary storage.

Upon entry to your ISP stem coded in C, the following conditions exist:

The machine state register (*MSR*) matches the state after the exception, conditioned as described in Chapter 3.8.
Interrupts are disabled and must remain so.
The stack pointer in register *r1* references the stack in effect at the time the interrupt exception occurred.
A register frame has been allocated on the stack.
All registers which C considers volatile have been saved and are free for use.
The return address is in register *lr*.
r3 contains the ISP parameter, if any.
All other registers must be preserved.

Upon return, *r3* contains a continuation directive.

The ISP stem must return a continuation directive, say *x*. If the ISP stem returns to the ISP root with *x = 0*, the ISP root will bypass the device ISP Handler. To force execution of the ISP Handler, the ISP stem must return with *x* not equal to *0*.

The ISP Handler can be written in assembly language or C. When writing in assembly language, you must adhere to the C register and parameter passing conventions defined for the particular implementation of C which you using.

Upon entry to your ISP Handler, the following conditions exist:

Interrupts are enabled but only quick interrupts are allowed.
lr contains the return address.
r3 contains the ISP parameter, if any.
The stack pointer register *r1* references the AMX Interrupt Stack.
All registers which C considers volatile are free for use.
All other registers must be preserved.

The following examples illustrate how simple an ISP stem and ISP Handler can be.

Assume that the ISP definition given to the AMX Configuration Builder for inclusion in the Target Parameter File is as follows:

The ISP root is given the public name *deviceisp*.
The assembly language ISP stem is named *devicestem*.
The ISP Handler is named *deviceh*.
The device interrupts on AMX vector number 18.
The ISP stem and ISP Handler do not require any parameter.

The assembly language ISP stem and C ISP Handler are coded as follows:

```
            .globl devicestem
devicestem:
            :
            Clear the source of the interrupt request
            :
#
# If the ISP Handler must be executed because of this interrupt
            addi  r31,r0,1
            blr
#
# Else
            addi  r31,r0,0
            blr


void CJ_CCPP deviceh(void)
{
   local variables, if required
   :
   Perform all operations necessary to complete the interrupt service.
   :
   }
```

Now assume that *dcbinfo* is some application device control block structure. Assume that *deviceXdcb* is a structure variable defined as follows:

```
struct dcbinfo deviceXdcb;
```

Then the ISP definition given to the AMX Configuration Builder for inclusion in the Target Parameter File can be as follows:

The ISP root is given the public name *dcb_isp*.
The ISP stem, coded in C, is named *dcb_stem*.
The ISP Handler is named *dcb_h*.
The device interrupts on AMX vector number 19.
deviceXdcb is the name of the public structure variable which contains information about the specific device.

The C ISP stem and ISP Handler are coded as follows:

```
int CJ_CCPP dcb_stem(struct dcbinfo *dcbp)
{
   local variables, if required
   :
   Use the device control block pointer dcbp to access structure
   variable deviceXdcb to determine device addresses.
   Clear the source of the interrupt request.
   :
   if (the ISP Handler must be executed because of this interrupt)
            return(1);

   return(0);                       /* Do not execute ISP Handler     */
   }


void CJ_CCPP dcb_h(struct dcbinfo *dcbp)
{
   local variables, if required
   :
   Use device control block pointer dcbp to access structure variable
   deviceXdcb to determine the required service actions.
   Perform all operations necessary to complete the interrupt service.
   :
   }
```

3.5 Nonconforming ISPs

The PowerPC family of processors provides an interrupt mechanism which permits the use of nonconforming ISPs within an AMX system. Since nonconforming ISPs bypass the AMX Interrupt Supervisor, they must execute with interrupts disabled and cannot make use of any AMX task synchronization or communication services.

In theory, the simplest nonconforming ISP is the exception handler which is executed by the PowerPC in response to the interrupt. Upon entry to such a nonconforming ISP, the processor state is dictated by the PowerPC. The processor is at supervisor level with all interrupts disabled (*EE = 0* in the machine state register). The save/restore registers *srr0* and *srr1* contain the machine context prior to the interrupt. All other registers must be preserved.

The nonconforming ISP MUST NOT enable interrupts.

The nonconforming ISP must dismiss the interrupt request and exit with an *rfi* instruction which restores the processor to its state prior to the interrupt and resumes execution at the point of interruption.

To create such a nonconforming ISP, you must code the exception handler in PowerPC assembly language and arrange for it to reside in the PowerPC Exception Vector Table.

Nonconforming User Exception Service Procedure

The simplest alternative to building your own exception handler is to let AMX install an AMX exception handler to dispatch directly to your nonconforming ISP. Such an ISP is just a custom user exception service procedure which services the interrupting device. In this case, the procedure can be written in assembly language or C as described in Chapter 3.1.

Multiplexed Nonconforming ISPs

If the device generating the nonconforming interrupt is one of several devices multiplexed through a single exception vector, a different approach must be taken. In this case, you must let AMX dispatch to your nonconforming device handler through the AMX Vector Table.

To do so, use the AMX Configuration Builder to define a nonconforming ISP which has an ISP stem coded in C (or assembly language) but no ISP Handler. Such an ISP is, by definition, a nonconforming ISP.

The conditions on entry to the nonconforming ISP stem are as defined in Chapter 3.4.

The ISP stem must return the value *0* to the ISP root since there can be no ISP Handler for a nonconforming ISP to execute.

The examples of Chapter 3.4, when treated as nonconforming ISPs, appear as follows:

```
            .globl devicestem
devicestem:
            :
            Clear the source of the interrupt request
            Complete all device service.
            :
            addi  r31,r0,0
            blr
```

```
int CJ_CCPP dcb_stem(struct dcbinfo *dcbp)
{
   local variables, if required
   :
   Use the device control block pointer dcbp to access structure
   variable deviceXdcb to determine device addresses.
   Clear the source of the interrupt request.
   Complete all device service.
   :
   return(0);
   }
```

3.6 Quick Interrupts

The AMX PPC32 kernel does not disable interrupts to protect its private critical regions of code. When AMX enters a critical region, it allows an ISP stem to service an interrupt but defers any request by the ISP stem to execute the corresponding ISP Handler. The ISP stem's request is added to the end of the AMX Handler Request Queue. When AMX leaves its critical region, it executes each of the deferred ISP Handler requests in the order in which they occurred.

AMX always executes an ISP Handler with interrupts enabled. Hence, while an ISP Handler is executing in response to one interrupt, an ISP stem is allowed to service an interrupt. However, once again AMX defers any request by the ISP stem to execute the corresponding ISP Handler. When AMX completes execution of the interrupted ISP Handler, it executes each of the remaining deferred ISP Handler requests in the order in which they occurred.

Interrupts which occur when AMX is in a critical region or while AMX is executing an ISP Handler are called **Quick Interrupts** because only the ISP stem is allowed to execute in response to the interrupt.

Interrupt State in AMX Procedures

AMX Procedures are documented in Chapter 18 (Section 3) of the AMX User's Guide. In each procedure description, the effect of the procedure on the state of the interrupt system is defined using the following legend.

■ Disabled ■ Enabled ■ Restored
(Not in ISP)

Interrupts are considered enabled or disabled according to the state of the interrupt enable flag (`EE`) in the current machine state register (`MSR`) as defined in Chapter 3.1. However, AMX PPC32 does not actually disable interrupts to close the critical sections of code within its procedures. Instead, AMX leaves the interrupt state unaltered but allows only quick interrupts during its critical operations. For AMX PPC32, the interrupt state legend is to be interpreted as follows:

D	E	R	Effect on Interrupts
□	□	□	Untouched
■	□	□	Disabled and left disabled upon return
□	■	□	Enabled and left enabled upon return
■	■	□	Untouched during critical sections but enabled upon return
■	□	■	Untouched. If interrupts are enabled upon entry, only quick interrupts will be allowed during critical sections.
■	■	■	Never disabled, possibly briefly enabled and then, prior to return, restored to the state in effect upon entry to the procedure. If interrupts are enabled upon entry, only quick interrupts will be allowed during critical sections.

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

3.7 Processor Vector Initialization

The PowerPC processor unconditionally jumps to a predetermined memory address in its Exception Vector Table whenever a particular exception occurs. The code located at that address is called an exception handler.

The Exception Vector Table must be located in RAM. AMX will install exception handler code fragments into the table for the exceptions which AMX has been directed to control. Your Target Parameter File (see Chapter 4) defines the exception vectors which AMX will control.

If you intend to locate the Exception Vector Table in ROM, then you must take responsibility for initializing all of the entries in the table. In particular, you must embed code fragments in the table for each of the exceptions controlled by AMX.

Unfortunately, there is no simple way to create such a ROMable copy of the Exception Vector Table. The image is sparse with code fragments at intervals of *0x0100*. For processors such as the IBM PPC403, the table is further complicated by the need for short code fragments to accommodate the timer interrupts at vector offsets *0x10n0*.

Many of the exception handlers must eventually call service procedures which reside at various locations in your AMX application load image. Getting such a complex code image into a ROM will therefore depend on the linker, locator and ROM burning tools available to you.

For these reasons, an AMX load module does not include a directly ROMable image of the Exception Vector Table.

If you must commit the Exception Vector Table to ROM, proceed as follows. Allocate a region of RAM for use by AMX as a shadow Exception Vector Table. Assume that this RAM region is at memory address *EVBASE*. The least significant 8 bits of *EVBASE* must be *0*.

If the exception at vector offset *VOFS* is defined in your Target Parameter File to be under AMX control, then in your ROM table, generate code at offset *VOFS* to branch to *EVBASE+VOFS* with no change in the machine state.

Use the AMX Configuration Builder to edit your Target Parameter File to let AMX know that a shadow Exception Vector Table is required. On the EVT Service properties page (see Chapter 4.2), declare that the Exception Vector Table is to be shadowed and enter the numeric equivalent of *EVBASE* as the Base address of the table.

Before you launch AMX, your main program must select your ROM Exception Vector Table. To do so, you must properly set the *IP* bit in the machine state register (*MSR*). If necessary, you must also program the PPC403 Exception Vector Prefix Register (*EVPR*), the MPC8560 or PPC440 Interrupt Vector Prefix Register (*IVPR*) or the MPC602 Interrupt Base Register (*IBR*) to reference your ROM Exception Vector Table.

Saving Exception Vectors

When AMX is launched, it does not save a copy of the Exception Vector Table. AMX does not save the original content of the PPC403 Exception Vector Prefix Register (*EVPR*), the MPC8560 or PPC440 Interrupt Vector Prefix Register (*IVPR*) or the MPC602 Interrupt Base Register (*IBR*).

If you intend to launch AMX temporarily, your main program should save a copy of the Exception Vector Table for subsequent restoral upon return from AMX. If necessary, read and preserve a copy of the *EVPR*, *IVPR* or *IBR*.

When AMX returns to your main program, the *IP* bit in the machine state register (*MSR*) will be restored to its state prior to launching AMX. Note that, since the decrementer exception cannot be disabled, AMX always plugs the decrementer exception vector (vector offset *0x0900*) with an *rfi* instruction to prevent catastrophic failure on exit from AMX. It is the responsibility of your AMX application to disable all other interrupt sources before shutting down AMX.

3.8 MSR in Exception Handlers

When an exception occurs, the PowerPC copies the current machine state register (*MSR*) to save/restore register *SRR1* and then updates the *MSR* in an exception dependent fashion. Because of this update, some of the configuration parameters in the *MSR* may actually be altered leaving memory or devices inaccessible by the exception handler.

To overcome this constraint, your Target Parameter File (see Chapter 4.2) can be used to define which, if any, of the control bits in the *MSR* are to be restored by the AMX exception handler to their state prior to the exception.

For example, if an alternate *MSR* value of *0x00002030* is provided in your Target Parameter File, all AMX exception handlers will restore the three control bits (*DR*, *IR* and *FP*) to their state prior to the exception. All other bits in the *MSR* will remain in the state dictated by the PowerPC exception.

> Warning!
>
> If the PowerPC Memory Management Unit (MMU) is used and memory caching is to be enabled, extreme care must be taken to ensure that the *MSR* register content is not corrupted when servicing exceptions, leading to catastrophic failure. See Appendix D.4 for details.

3.9 Decrementer Interrupt

Most members of the PowerPC family include a decrementer register which is automatically decremented at a fixed frequency, usually determined by a subdivision of the bus frequency. If interrupts are enabled (*EE = 1* in the *MSR*), the decrementer unconditionally generates an exception through exception vector *0x0900* whenever the most significant bit (bit *0*) of the register value changes from *0* to *1*.

The AMX Configuration Builder allows you to define how the decrementer exception is to be used in your application. Use the builder to edit your Target Parameter File (See Chapter 4.2). On the Exceptions page, pick the decrementer from the list of exceptions and choose the way it is to be used. Then, in the fields presented by the builder, if any, enter the additional information needed by AMX to support your choice.

Decrementer as an AMX Clock

The decrementer can be used as the source of AMX clock ticks. To do so, you must choose the AMX Clock decrementer option. Define the vector number in the AMX Vector Table through which the decrementer interrupt will be vectored. You must also provide the 32-bit value used to define the AMX clock interrupt frequency (See Chapter 5.3.1).

The decrementer frequency can also be dynamically adjusted at run time. To do so, configure the decrementer value to be *0*. Your *main()* program must then install the real decrementer value into *long* variable *cjcfdecrvalue* prior to launching AMX. Thereafter, any change which you make to the value of variable *cjcfdecrvalue* will take effect at the next decrementer exception.

Decrementer Interrupts

You can use the decrementer exception as a special source of interrupts by choosing the Interrupt decrementer option.

To treat the decrementer exception as a single device interrupt, set the number of devices to one and enter the AMX vector number which you wish to assign to the device.

You can also treat the decrementer exception as a multiplexed device interrupt assigned to the set of *VNCOUNT* vectors beginning at AMX vector number *VNBASE*. To do so, set the number of devices to *VNCOUNT* and enter *VNBASE* as the first AMX vector number which you wish to assign to the collection of devices.

In this case, you must also enter the name of your Interrupt Identification Procedure (IIP) which AMX will call to identify the device requiring service. Your IIP can restart the decrementer by loading a new value into the decrementer register.

You must define a conforming AMX Interrupt Service Procedure (ISP) for the artificial device (or devices) which your decrementer represents. Each ISP is defined as described in Chapters 3.4 and 4.3.

Custom Decrementer Exception

You can also use the decrementer exception for you own purposes. To install an AMX exception handler which will call your custom user exception service procedure, simply declare that you wish to use the decrementer as a User exception. Enter the name of your procedure and the parameter, if any, which you wish it to receive. Your user exception service procedure can restart the decrementer by loading a new value, possibly your defined parameter, into the decrementer register.

Ignoring the Decrementer

The decrementer unconditionally generates an exception through exception vector *0x0900* whenever bit *0* of the decrementer register value changes from *0* to *1* unless interrupts are disabled (*EE = 0* in the *MSR*). Since the decrementer exception cannot be inhibited, it must be "plugged" with an *rfi* instruction in its entry in the Exception Vector Table.

If you have no use for the decrementer, choose the Plugged decrementer option and AMX will plug the decrementer exception vector with an *rfi* instruction.

In some cases, you may wish to allow a decrementer exception handler which is in place prior to launching AMX to retain control of the decrementer exception. To do so, choose the Unaltered decrementer option and AMX will leave the decrementer exception untouched.

On MPC555x, MPC85xx and PPC440 targets, the decrementer interrupt is enabled by setting the Decrementer Interrupt Enable (*DIE*) bit of the Timer Control Register (*TCR*) in addition to setting the *EE* bit in the *MSR*. Since the decrementer interrupt can be masked independent of the *EE* bit, the decrementer exception vector does not need to be plugged with an *rfi* instruction for the system to operate correctly. In such systems, the decrementer exception vector will always be unaltered by AMX unless the AMX target configuration specifies otherwise.

MPC601 Decrementer Restriction

Most PowerPC implementations set the decrementer register to *0xFFFFFFFF* following a hardware reset. The processor bootstrap code is thereby given a long interval to complete its initialization of the Exception Vector Table before the first decrementer exception can occur.

Unfortunately, the MPC601 sets the decrementer register to *0* following a hardware reset. As a result, a decrementer exception will occur whenever interrupts are first enabled (*MSR* bit *EE = 1*). Your MPC601 startup code must accommodate this characteristic by installing an *rfi* instruction (or suitable exception handler) at vector offset *0x0900* in the Exception Vector Table before enabling interrupts. The ROM monitor on most evaluation boards will meet this requirement.

If your startup code runs with interrupts disabled and you launch AMX with interrupts disabled, the pending MPC601 decrementer exception will occur as soon as AMX enables interrupts. It is therefore imperative that you install a valid decrementer exception handler in the current MPC601 Exception Vector Table prior to launching AMX.

This page left blank intentionally.

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, *CJ382CM.EXE*. 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.

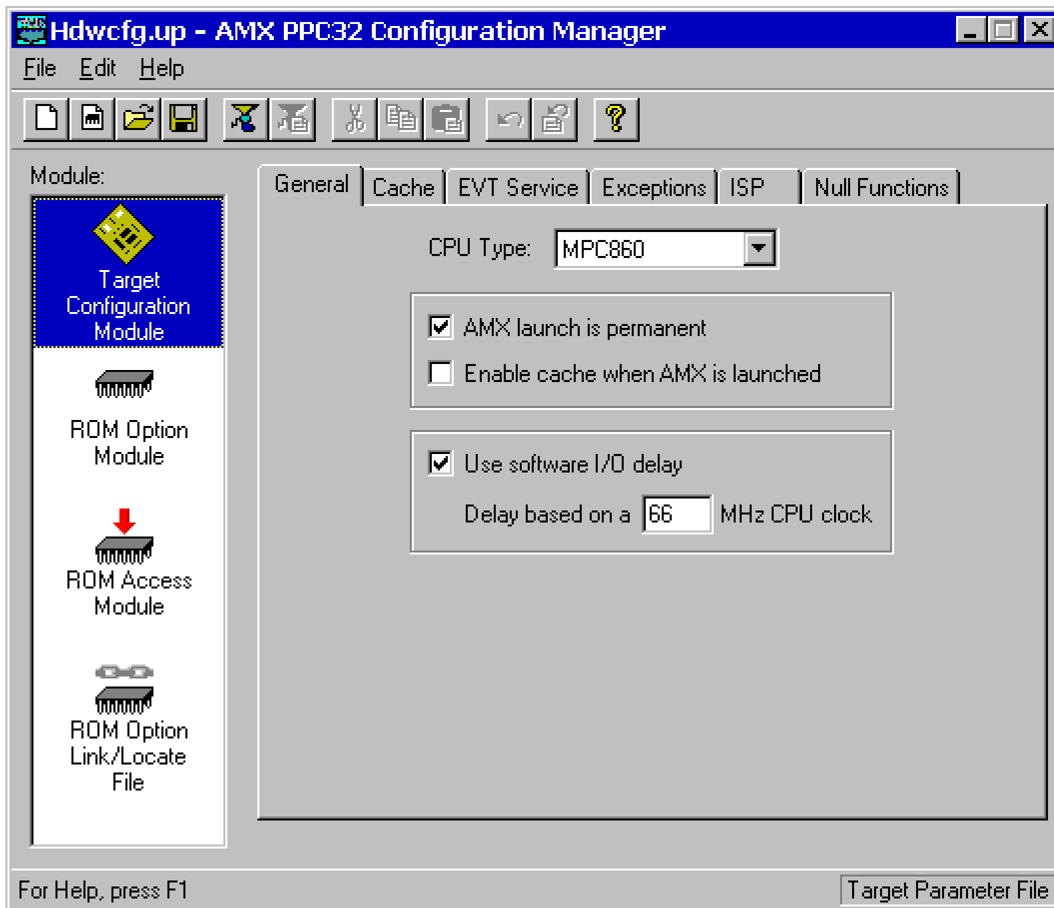


Figure 4.1-1 Configuration Manager Screen Layout

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. The currently supported list of processors includes:

401, 401A1, 401X2, 401B3, 401GF,	{IBM PPC401 or equivalent}
403, 403GA, 403GB, 403GC, 403GCX,	{IBM PPC403 or equivalent}
405, 405B3, 405GP,	{IBM PPC405 or equivalent}
440,	{IBM PPC440 or equivalent}
505, 509, 555,	{Motorola MPC5xx or equivalent}
5553, 5554,	{Freescale MPC555x or equivalent}
601, 602, 603, 604,	{Motorola MPC60x or equivalent}
603e, 5200, 8240, 8250, 8260, 8280, 8349,	{Motorola MPC603e or equivalent}
740, 750, 7400,	{Motorola MPC7xx or equivalent}
801, 821, 823, 850, 855, 860,	{Motorola MPC8xx or equivalent}
8540, 8560	{Motorola MPC85xx or equivalent}

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

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 indicated by field CPU Type has no cache control, leave this box unchecked.

If the processor indicated by field CPU Type has cache control, then, before launching AMX, you must initialize the Memory Management Unit (MMU) to condition the instruction and data block address translation registers (*IBATn* and *DBATn*) to meet the caching requirements of your system. AMX procedures *cjcfhwbatrd* and *cjcfhwbatwr* can be used to read and modify the *xBATn* registers.

You must also condition the machine state register (*MSR*) to enable/disable block address translation for instructions (*MSR* bit *IR*) and/or data (*MSR* bit *DR*). AMX procedures *cjcfflagrd* and *cjcfflagwr* can be used to read and modify the *MSR*.

You may also have to condition the hardware implementation register (*HID0*) to globally enable/disable the instruction cache (*HID0* bit *ICE*) and/or data cache (*HID0* bit *DCE*). AMX procedure *cjcfmodhid0* can be used to read and modify the *HID0* register.

The example provided in the description of AMX procedure *cjcfhwbatrd* illustrates the cache setup procedure for the Ultra 603 board.

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.

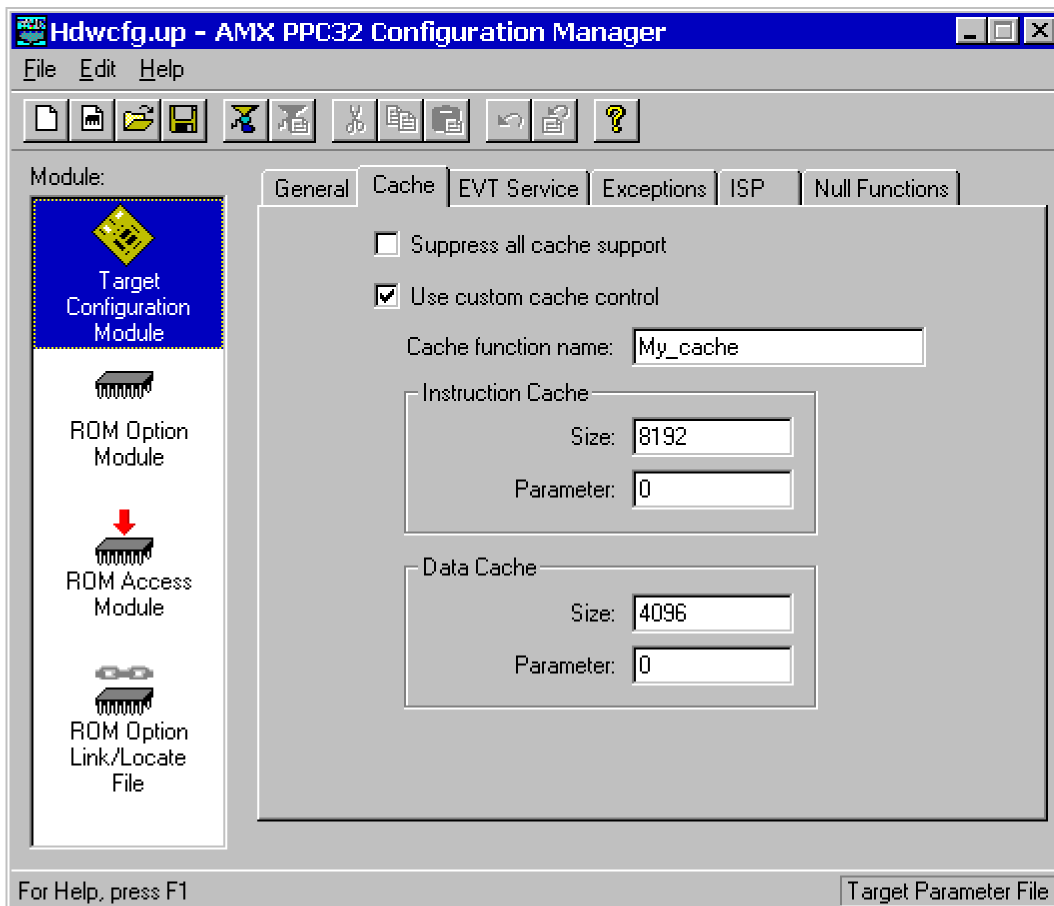
Leave this box unchecked if you want the I/O delay procedure *cjcfhwdelay* to produce no delay beyond that inherent in the procedure call and return.

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 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

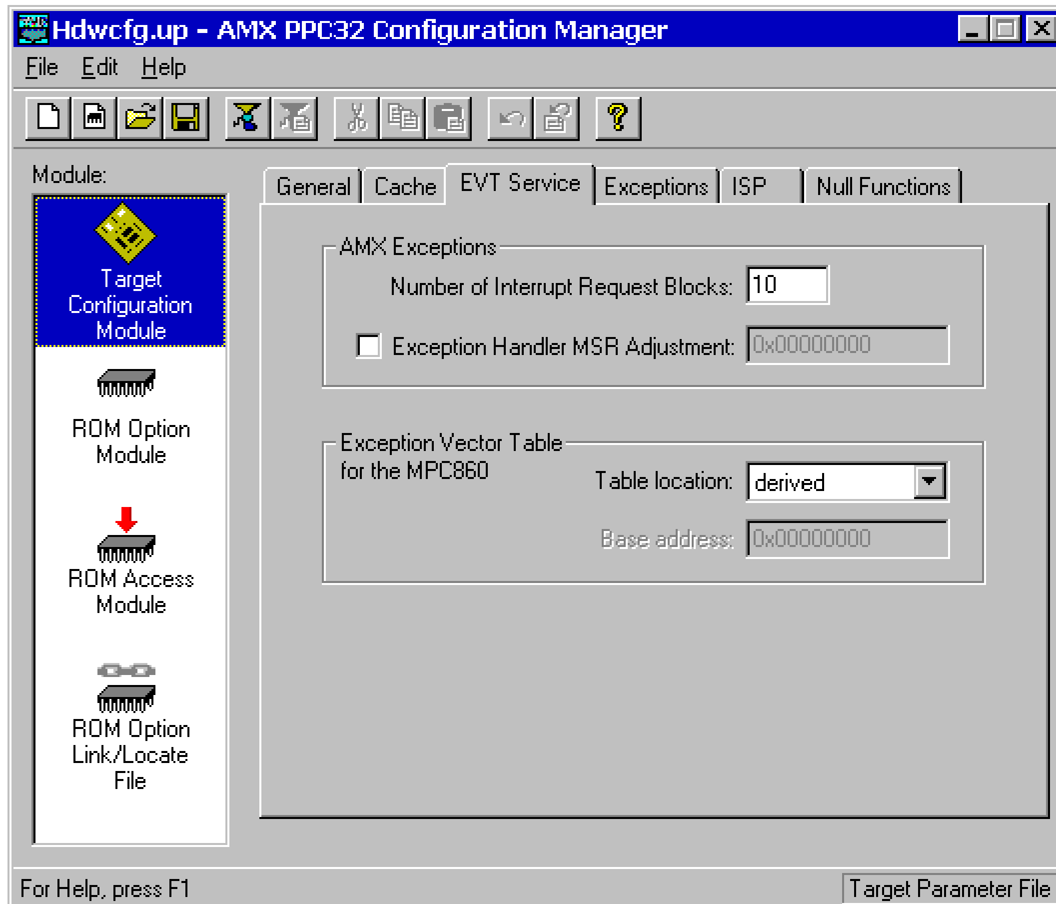
If you provide your own custom cache control function, the interpretation of the four cache control parameters must be as follows.

The parameter `icsize` must define the total size, in bytes, of the instruction cache. The least significant 16 bits of parameter `icparam` must define the number of bytes in each instruction cache block (cache line). The upper 16 bits are free for interpretation by your cache control function.

The parameter `dcsize` must define the total size, in bytes, of the data cache. The least significant 16 bits of parameter `dcparam` must define the number of bytes in each data cache block (cache line). The upper 16 bits are free for interpretation by your cache control function.

Exception Service Parameters

The Target Configuration Module includes parameters to control how AMX services exceptions. These parameters are entered in the Exception Service window. The layout of the window is shown below.



Interrupt Request Blocks

AMX uses an Interrupt Request Block to defer execution of any ISP Handler while AMX is in any critical region of code. This parameter defines the number of Interrupt Request Blocks needed to accommodate the worst case interrupt deferral which may occur in your application.

For most applications, set the number of Interrupt Request Blocks to twice the number of interrupt sources. For example, if you use the PowerPC decrementer as an AMX clock and have two other interrupting devices sharing the external interrupt exception, set the number to 6.

If an Interrupt Request Block is not available at the time that AMX must defer the request of an ISP stem to execute its ISP Handler, AMX will generate a fatal exit with fatal error code *CJ_FENOIRB*.

Exception Handler MSR Adjustment

When an exception occurs, the PowerPC copies the current machine state register (*MSR*) to save/restore register *SRR1* and then updates the *MSR* in an exception dependent fashion. Because of this update, some of the configuration parameters in the *MSR* may actually be altered leaving memory or devices inaccessible by the exception handler.

To overcome this constraint, you can check this box and enter an adjustment parameter to define which of the control bits in the *MSR* are to be restored by the AMX exception handler to their state prior to the exception. The *MSR* control bits are identified by setting the corresponding bits in this parameter.

For example, if the *MSR* adjustment parameter is set to *0x00002030*, then all AMX exception handlers will restore the three control bits (*DR*, *IR* and *FP*) to their state prior to the exception. All other bits in the *MSR* will remain in the state dictated by the PowerPC exception.

If you do not need to restore the state of any *MSR* control bits within AMX exception handlers, leave this box unchecked.

Exception Vector Table

AMX must be able to determine the location of the PowerPC Exception Vector Table in order to install into the table the code fragments necessary to service the exceptions. The Exception Vector Table is located at memory address 0 for most PowerPC implementations. However, for some PowerPCs, the location of the table may be adjusted.

The Table location pull down list lets you choose how the address of the Exception Vector Table is to be determined. The table address can be derived, adjustable or shadowed.

EVT Location Derived by AMX

If you indicate that the Exception Vector Table location is to be **derived**, AMX will derive the base address of the processor's Exception Vector Table at launch time according to the processor type specified by parameter CPU Type on the General Parameters page. The Base address parameter will not be used.

For processors such as the **MPC50x, MPC5200, MPC601, MPC603, MPC604, MPC7xx, MPC8xx and MPC82xx** which adhere to the PowerPC Architecture specification, AMX uses the *IP* bit in the machine state register (*MSR*) to determine the address of the Exception Vector Table. If *IP = 0,* the table is at address *0*; otherwise the table is at address *0xFFF00000*.

For processors such as the **PPC403 and PPC405**, the Exception Vector Prefix Register (*EVPR*) determines the base address of the Exception Vector Table. AMX will read the *EVPR* to determine the base address.

For processors based on the PowerPC Book E specification, such as the **MPC555x, MPC85xx and PPC440** families, the Interrupt Vector Prefix Register (*IVPR*) determines the base address of the Exception Vector Table. AMX will read the *IVPR* to determine the base address.

For the **MPC602**, the *IP* bit in the machine state register (*MSR*) and the Interrupt Base Register (*IBR*) determine the base address of the Exception Vector Table. If *IP = 1*, the table is at address *0xFFF00000*. If *IP = 0,* critical exception vectors are unconditionally at address *0* and all others are at the base address determined by the content of the Interrupt Base Register (*IBR*). AMX will read the state of the *IP* bit and the content of the *IBR*, if necessary, to derive the base address.

EVT Location Adjustable

If you indicate that the Exception Vector Table location is to be **adjustable**, AMX will set the base address of the processor's Exception Vector Table at launch time to the value you select or enter in the Base address field. The allowable base address values are determined by the processor type specified by parameter CPU Type on the General Parameters page.

For processors such as the **MPC50x, MPC5200, MPC601, MPC603, MPC604, MPC7xx, MPC8xx and MPC82xx** which adhere to the PowerPC Architecture specification, there are only two possible locations for the Exception Vector Table. You must choose the address from the pull down list presented in the Base address field. AMX sets *IP = 0* in the machine state register (*MSR*) if the table is to be at address *0*. AMX sets *IP = 1* if the table is to be at address *0xFFF00000*.

For processors such as the **PPC403 and PPC405**, the Exception Vector Prefix Register (*EVPR*) determines the base address of the Exception Vector Table. AMX will copy the value from the Base address field into the *EVPR*, thereby establishing the base address of your choice. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

For processors based on the PowerPC Book E specification, such as the **MPC555x, MPC85xx and PPC440** families, the Interrupt Vector Prefix Register (*IVPR*) determines the base address of the Exception Vector Table. AMX will copy the value from the Base address field into the *IVPR*, thereby establishing the base address of your choice. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

For the **MPC602**, the *IP* bit in the machine state register (*MSR*) and the Interrupt Base Register (*IBR*) determine the base address of the Exception Vector Table. If you enter a Base address value of *0xFFF00000*, AMX will set the *IP* bit to *1* thereby forcing the base address to be *0xFFF00000*. If you enter any other Base address value, AMX will set the *IP* bit to *0* and copy that value into the *IBR*, thereby establishing your value as the base address for all but critical exceptions. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

EVT Located in ROM

If your Exception Vector Table is located in ROM, AMX cannot install its exception handlers into the table. You must provide a **shadowed** Exception Vector Table in RAM. This procedure is described in Chapter 3.7. In the Base address field, enter the memory address at which the shadow table is to be located. Note that a valid base address must have the least significant 8 bits of the address set to *0*.

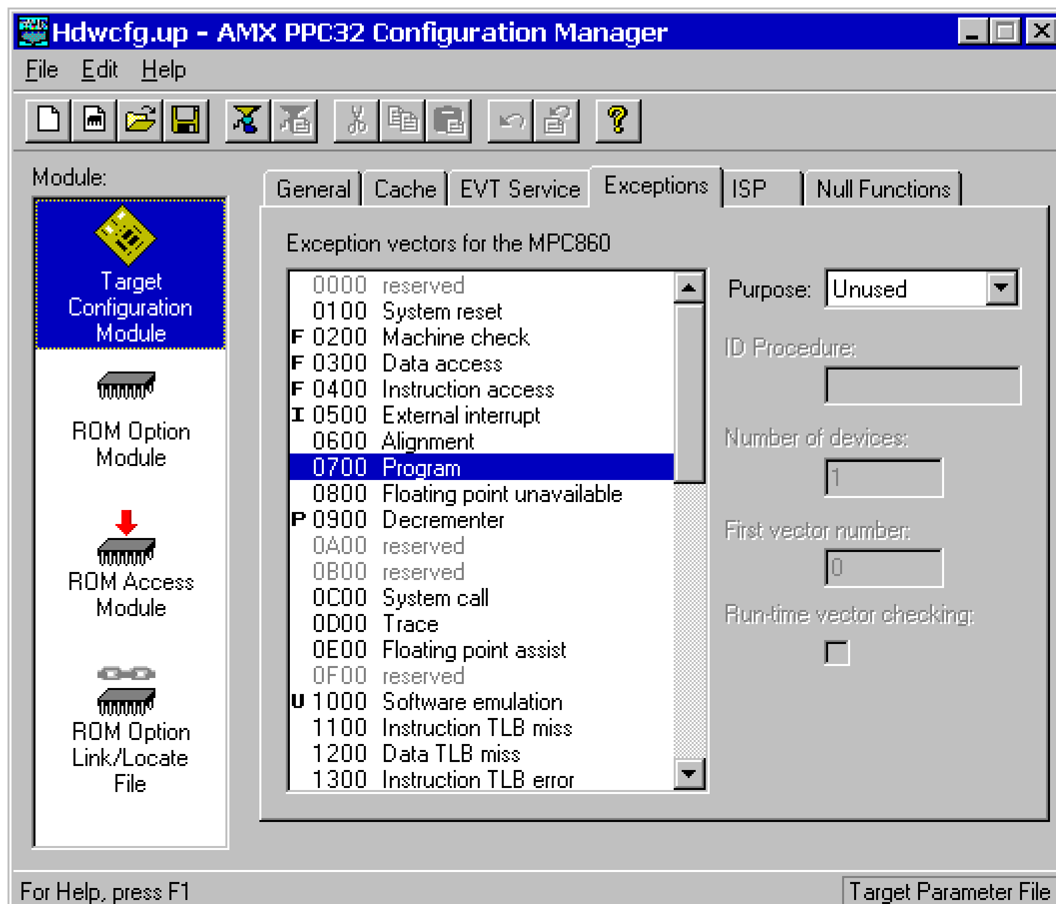
Exception Definitions

The Target Configuration Module defines the processor exceptions which are to be serviced by AMX. These exceptions are specified by you by selections in the Exception Definition window. The layout of the window is shown below.

The PowerPC selected by the CPU Type on the General Parameters page is identified at the top of the Exception Definition window. The exception list shows all exceptions from *0x0000* to *0x2F00*. The exceptions present in the PowerPC of interest are identified by name. All others are dimmed and shown as reserved. To the left of each exception is a single character identifying the manner in which the exception will be serviced by AMX.

blank	Unused (untouched by AMX)	
I	Interrupt	
U	User exception	
F	Fatal	
P	Plugged with *RFI*	(decrementer only)
C	AMX clock	(decrementer only)

Any reserved exception which you have chosen to use for some purpose will be highlighted in the error color set using the Edit, Preferences... dialog.



Vector Purpose

To establish the purpose of a particular exception in the Exception Vector Table, click on that exception in the exception list. Then, from the pull down list labeled Purpose, choose how you wish to use the exception.

Exceptions that are marked as **Unused** will not be touched by AMX. It is up to your application to take full responsibility for the exception.

Exceptions that are marked as **Fatal** will be serviced by AMX and treated as fatal as described in Chapter 3.1.

If one or more internal or external devices generate interrupts through an exception vector, the exception must be marked for **Interrupt** use. Such exceptions will be serviced by the AMX Interrupt Supervisor. The interrupt exception details must be defined as described later in this chapter.

Each exception that is marked as a **User Exception** will be serviced by an AMX exception handler which will call your custom user exception service procedure. The AMX exception handler copes with the assembly language details of servicing the exception leaving your procedure free to deal only with the reason for the exception. The user exception details must be defined as described later in this chapter.

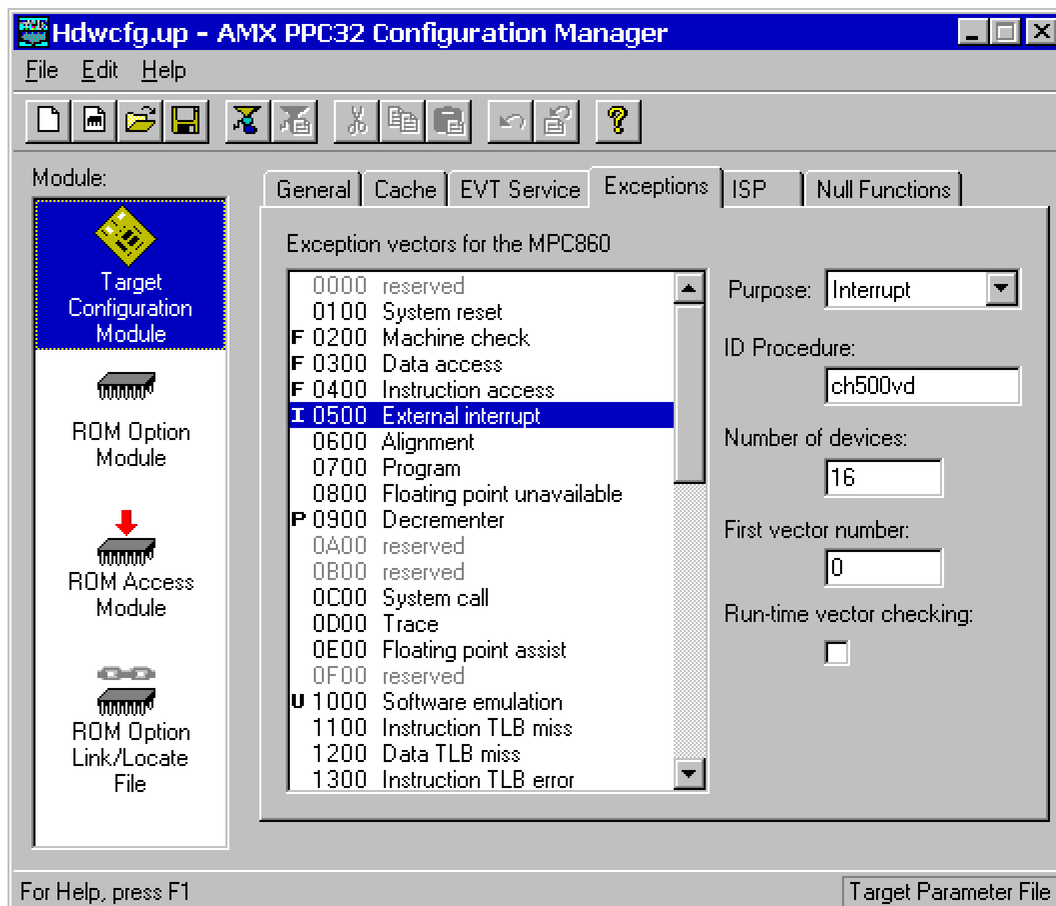
Note

The **decrementer** exception is a special case which is described later in this chapter.

Interrupt Exceptions

The example illustrated below shows the PowerPC external interrupt exception (vector `0x0500`) being defined for use as a source of AMX device interrupts.

Note that for each of the interrupting devices, you must also define an Interrupt Service Procedure (ISP) as described in Chapter 4.3.



Dedicated Device Interrupt

If a **single device** generates an interrupt through an exception vector, fill in the exception definition as follows. Set the Number of devices to one (1). In the field labeled First vector number, enter the vector number in the AMX Vector Table which you wish to reserve for the device. A warning indication will appear if the vector number you enter is already used by some other interrupt exception.

Since there is only one device, leave the ID procedure field blank (empty) and leave the Run-time vector checking box unchecked. You will not be able to leave the Exception Definition window if these unused fields are improperly filled. An error message describing the fault will appear in the status bar.

Multiplexed Device Interrupts

If **multiple devices** generate interrupts through the interrupt exception, fill in the 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 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 interrupt exception. Enter the base vector number of the block of *n* AMX vectors which you wish to assign to these devices. A warning indication will appear if the vector number range you assign to this exception overlaps with vectors already used by some other interrupt exception.

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.

Note that the size of the AMX Vector Table will be determined by the highest vector number which you allocate to any interrupt exception.

Run-time Vector Checking

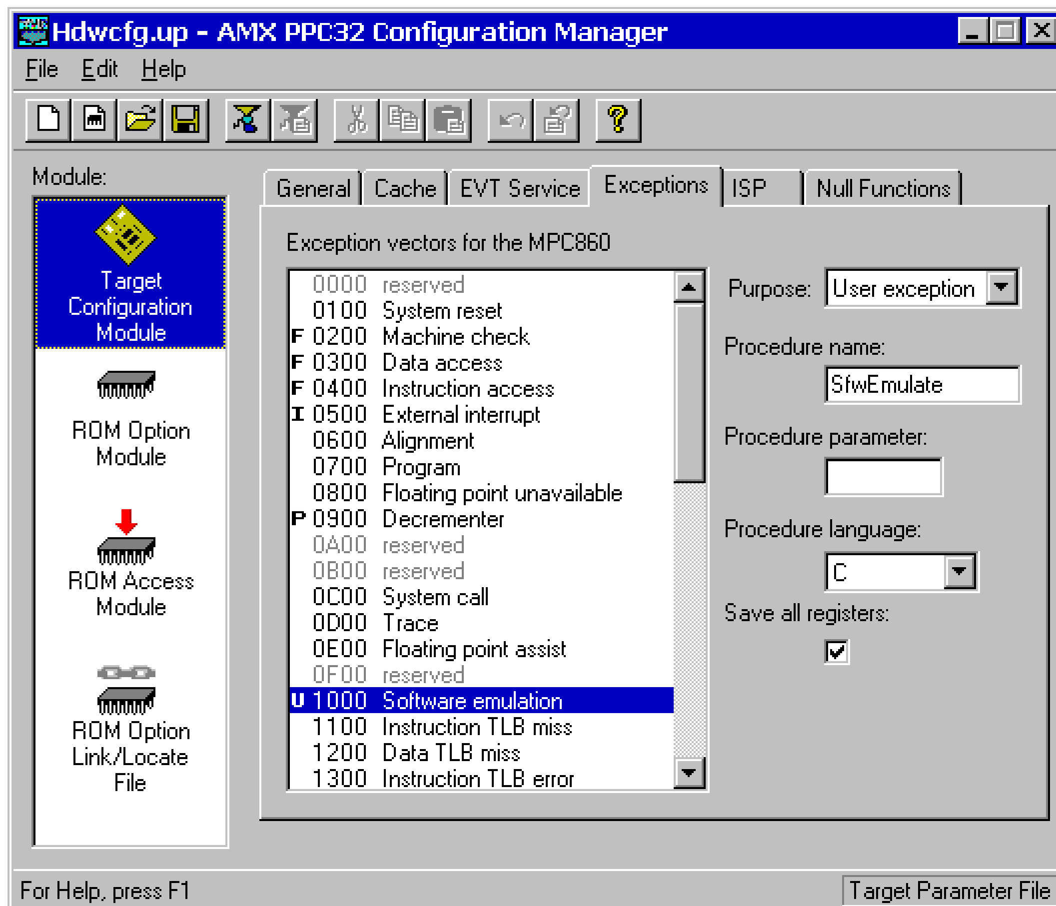
The AMX Interrupt Supervisor can check that the derived vector number lies within the allowable range in the AMX Vector Table. If it does not, AMX calls the Fatal Exception Handler indicating that an unidentified interrupt (exception variety *CJ_PRXVUIR*) occurred. The vector offset of the offending interrupt exception is passed to the handler as a parameter.

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.

User Exceptions

Each exception that is marked as a User exception will be serviced by an AMX exception handler which will call your custom user exception service procedure. The AMX exception handler copes with the assembly language details of servicing the exception leaving your procedure free to deal only with the reason for the exception.

The example illustrated below shows the PowerPC software emulation exception (vector *0x1000*) being defined as a user exception.



Procedure Name

Enter the name of your user exception service procedure. This is the name of the procedure which will be called by the AMX exception handler once the machine state has been preserved according to the conditions which existed at the time of the exception. Your procedure must abide by the calling conventions described in Chapter 3.1.

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

Procedure Parameter

Your user exception service procedure can receive a custom integer parameter in the range *0* to *0x000FFFFF*. The parameter will be merged with the exception variety code *CJ_PRXVUSR* before being passed to your procedure. Enter the required 20-bit hexadecimal numeric value into the Procedure parameter field.

If your procedure has no need for a custom parameter, leave the Procedure parameter field blank (empty). In this case, the parameter received by your procedure will simply be the exception variety code *CJ_PRXVUSR* merged with the vector offset of the exception being serviced.

Procedure Language

Your user exception service procedure can be coded in C or assembly language. Identify the language in which your procedure is written by picking C or Assembly from the pull down list.

Save All Registers

If this box is left unchecked, the AMX exception handler will only save a few registers before it calls your user exception service procedure. If your procedure is coded in C, AMX only saves the registers which C considers to be alterable. These few registers are saved in an AMX register structure *cjxregs* which is then made accessible to your procedure.

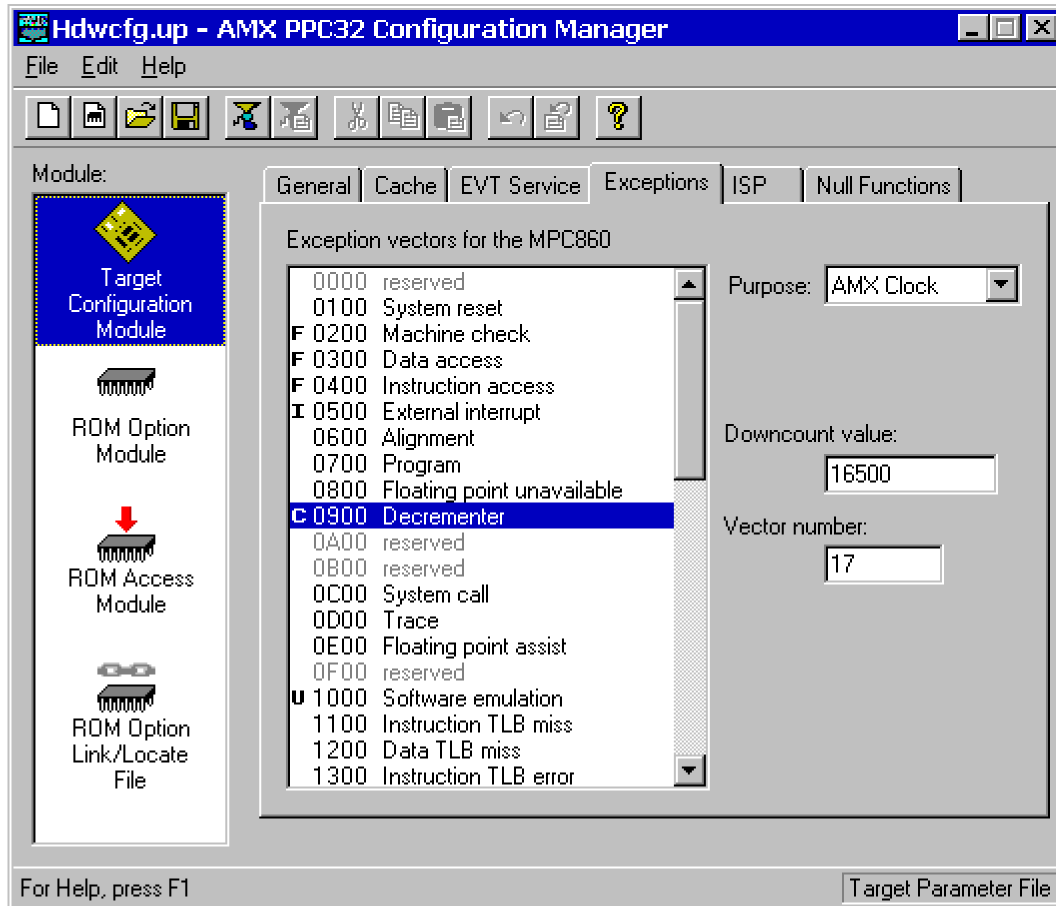
If your user exception service procedure requires access to the complete complement of registers as they existed prior to the exception, check this box. In this case, the AMX exception handler will save all registers in the register structure before calling your procedure.

Note that this option is not supported if your procedure is coded in assembly language. Such procedures have direct access to the full set of registers.

Decrementer Exception

The decrementer exception is a special case. Since the decrementer interrupt cannot be inhibited, it cannot be treated as unused. The exception will occur and must be serviced.

As the example illustrated below shows, the PowerPC decrementer interrupt (exception vector `0x0900`) can also be used as an AMX clock.



Vector Purpose

To establish the purpose of decrementer exception, click on that exception in the exception list. Then, from the pull down list labeled Purpose, choose how you wish to use the exception.

If you intend to service the decrementer exception yourself or wish to use the decrementer exception handler which was in place prior to launching AMX, select the **Unaltered** option to force AMX to leave the decrementer exception vector untouched.

Since the decrementer interrupt cannot be inhibited, it cannot be treated as unused. Instead, you can treat it as **Plugged**. AMX will insert an *rfi* instruction into the decrementer exception vector to force the decrementer interrupt to be ignored every time it occurs.

Although you can describe the decrementer exception as **Fatal**, doing so is somewhat meaningless since the interrupt, which cannot be inhibited, will occur at some time and cause a fatal condition.

The decrementer can be used as a source of AMX interrupts by defining it to be an **Interrupt** exception. Used in this fashion, the decrementer can be treated as a single device interrupt which occurs at intervals determined by the downcount value you load into the decrementer register. The decrementer can also be used to periodically poll a set of non-interrupting devices, thereby making them appear interrupt driven as far as your AMX application is concerned. The interrupt exception must be defined as described earlier in this chapter.

The decrementer can be serviced as a **User exception**. When used this way, your user exception service procedure becomes responsible for servicing the decrementer. The user exception must be defined as described earlier in this chapter.

Using the Decrementer as the AMX Clock

In order to use the AMX Decrementer Clock Driver as the source of your AMX clock tick, you must declare the decrementer exception to be an **AMX Clock**. Then follow the procedures described in Chapter 5.3.1 to include the AMX Decrementer Clock Driver as part of your AMX application.

Enter the **Downcount value** required to generate decrementer interrupts at the desired clock frequency. This value must be a positive, 32-bit value. The formula for deriving this value is presented in Chapter 5.3.1.

The decrementer frequency can also be dynamically adjusted at run time. To do so, configure the decrementer value to be *0*. Your *main()* program must then install the real decrementer downcount value into *long* variable *cjcfdecrvalue* prior to launching AMX. Thereafter, any change which you make to the value of variable *cjcfdecrvalue* will take effect at the next decrementer exception.

Enter the **Vector number** in the AMX Vector Table which you wish to reserve for the decrementer clock. A warning indication will appear if the vector number you enter is already used by some other interrupt exception.

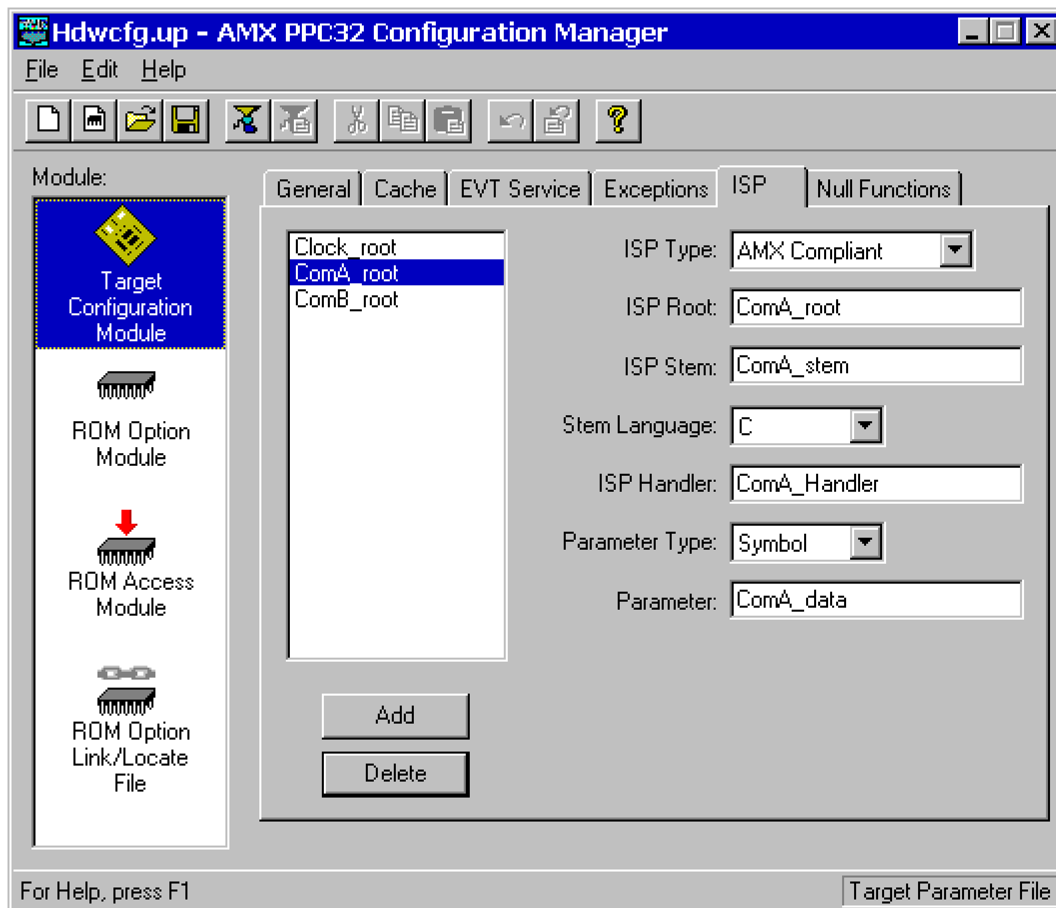
4.3 Interrupt Service Procedure (ISP) Definitions

Your Target Configuration Module must include a device ISP root for each interrupt device 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.



ISP Type

Most of your application ISPs will be **conforming AMX ISPs** which you define by choosing AMX Compliant from the pull down list. ISPs of this type are described in Chapter 3.4

You can also create a **nonconforming AMX ISP** which you define by choosing Nonconforming from the pull down list. ISPs of this type are described in Chapter 3.5

If you are using the decrementer as an AMX clock, there is no need to provide a clock ISP. On the Exceptions property page, simply declare the decrementer exception to be an AMX Clock.

Otherwise, 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 **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.

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.

ISP Stem

Enter the name of your device ISP stem which will clear the device interrupt request. This is the name of the procedure which will be called from the ISP root by the AMX exception handler once the interrupt source has been identified and the machine state preserved according to the conditions which existed at the time of the interrupt. The ISP stem cannot use AMX services. Your ISP stem must be coded as described in Chapter 3.4 or 3.5.

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

Stem Language

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

ISP Handler

Enter the name of your device ISP Handler which will complete service of the device. This is the name of the procedure which will be called by the AMX Interrupt Supervisor when so requested by the ISP stem. The ISP Handler can use AMX services. Your ISP Handler must be coded as described in Chapter 3.4.

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

Note that a nonconforming ISP has no ISP Handler. Consequently, the ISP Handler field is dimmed when you indicate that the ISP Type is Nonconforming.

ISP Parameter

Your ISP stem and ISP Handler can receive a 32-bit parameter every time they are called. The Parameter Type field is a pull down list used to identify what kind of parameter, if any, your procedures expect. If your ISP stem and ISP Handler have no need for a parameter, set the Parameter Type to **(none)**.

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

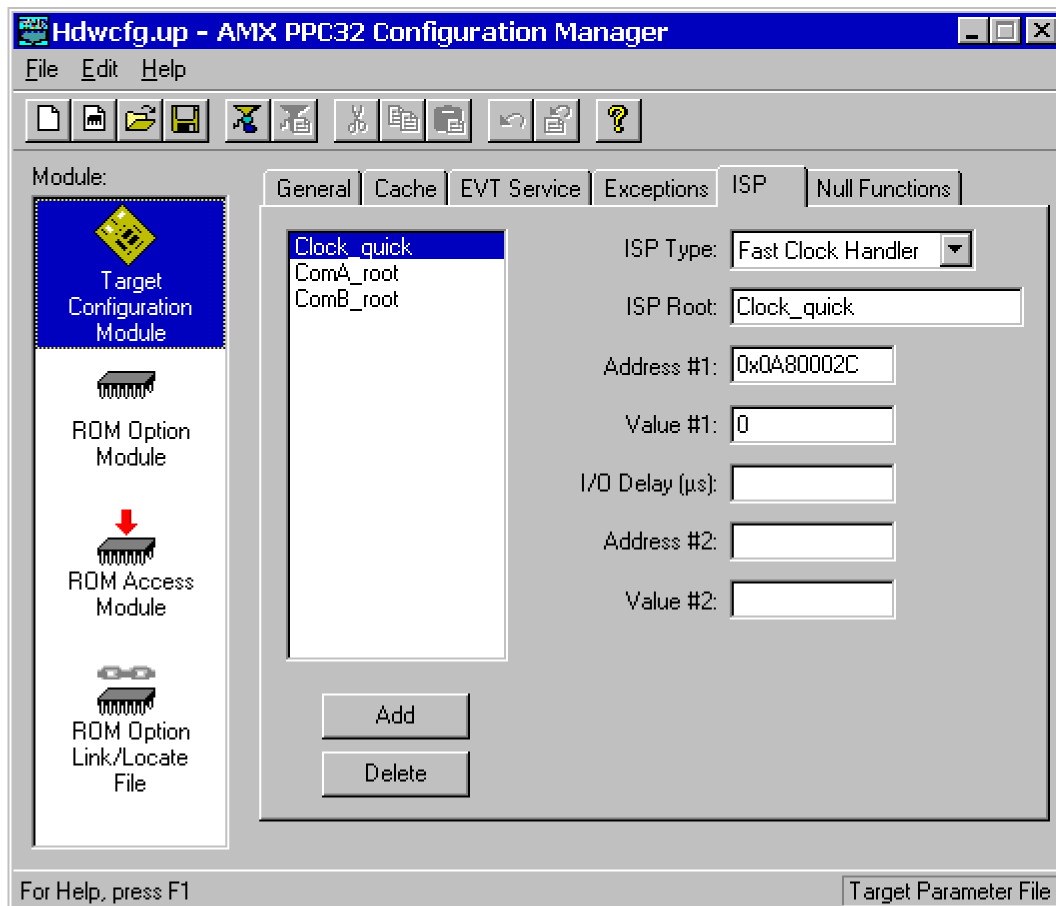
If your ISP stem and ISP Handler parameter must be a pointer to a variable or function, set the Parameter Type to **Symbol** and enter the name of the variable or function into the Parameter field. The parameter must be a text string giving the name of a public symbol (variable or function) defined in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

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 ISP 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 byte 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.



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 8-bit Value #1 to the device port at memory Address #1.
> Delay for the number of µs defined as I/O Delay (µs).
> Write 8-bit 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 a byte, is decoded by your target hardware as a reference to your clock device. Each value parameter is an 8-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 a byte 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 a byte be written to one device port, leave the I/O Delay field blank (empty).

4.5 Null Functions

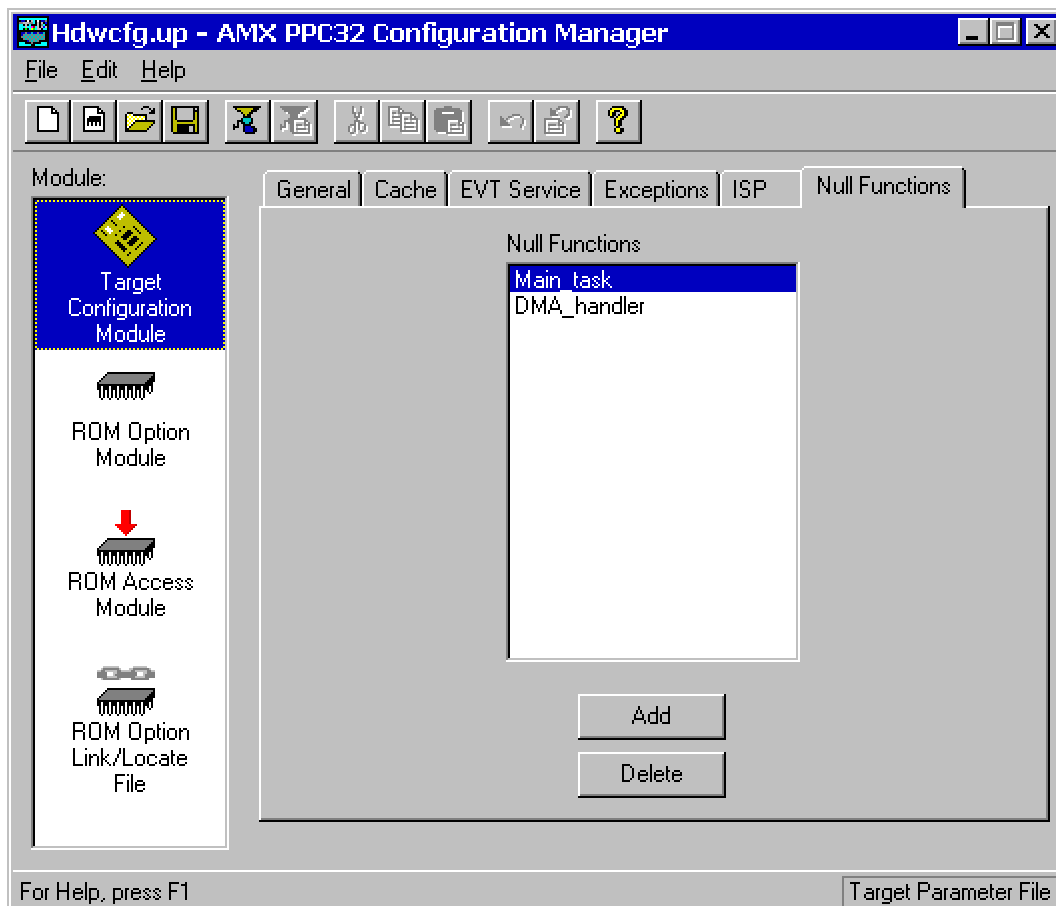
Occasionally, while developing an AMX application, it can be very convenient to be able to create software functions to satisfy your program link requirements without having to create the final version of these functions. For example, if your AMX System Configuration Module references a Restart Procedure 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.



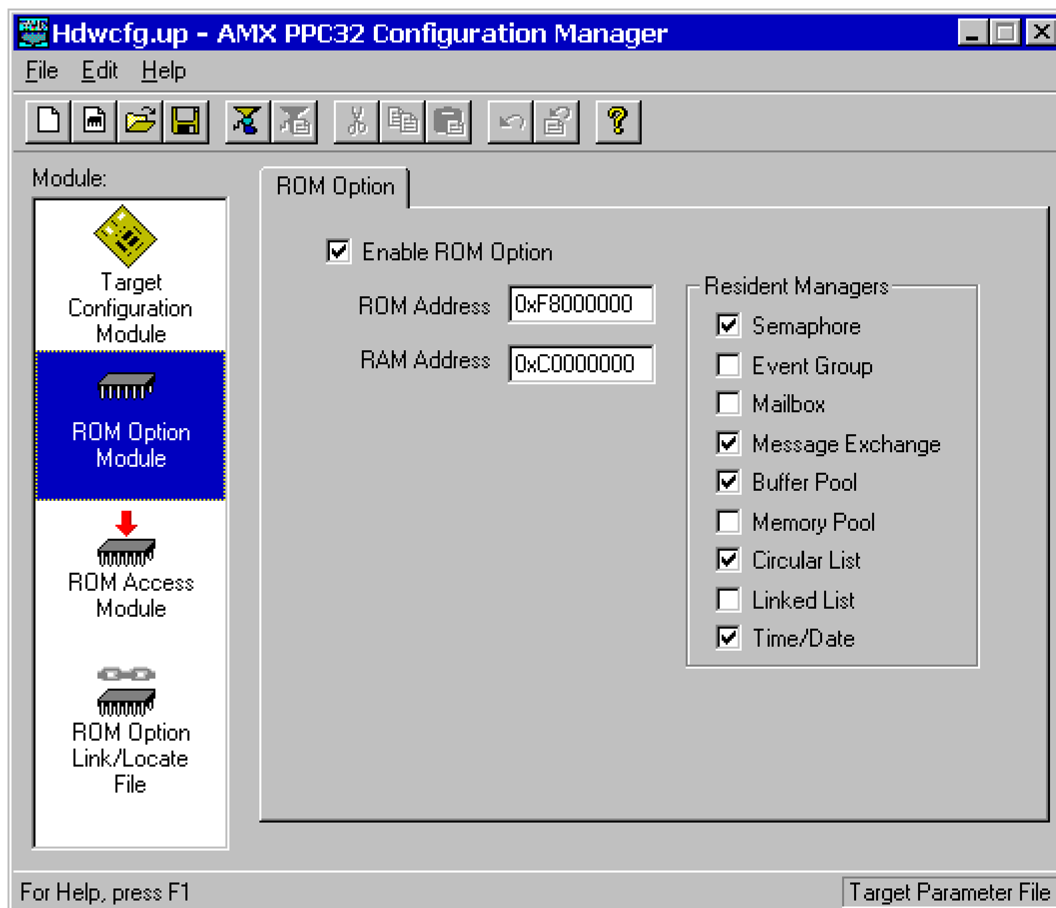
4.6 ROM Option Parameters

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 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

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.

This page left blank intentionally.

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

An external real-time clock used with the PowerPC processor will usually interrupt via the external interrupt exception. If the clock is one of several devices interrupting on the exception, your Interrupt Identification Procedure must sense that your clock is the device demanding service and provide the AMX exception handler with the interrupt identification number assigned by you to the clock. Once the AMX exception handler has determined that your clock is the interrupting device, it dispatches through its Vector Table to your clock ISP root.

The **clock ISP** consists of an ISP root, an ISP stem and an ISP Handler. The ISP root is called by the AMX exception handler in response to the clock interrupt request. The ISP root calls the ISP stem to dismiss the clock interrupt request and then, if so requested, posts a request for the execution of the clock ISP Handler. 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 stem or ISP 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.

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) 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 `0x80000040`.
The clock interrupt is generated through the external interrupt exception vector `0x0500`.
No other devices interrupt through exception vector `0x0500`.
The clock is assigned 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.

A conforming clock ISP for such a device could be coded as follows. Note that the ISP stem is coded in assembly language. No ISP Handler is required.

```
          .globl   clockstem
clockstem:
#
# Receives r31 = ISP parameter = A(clock base)
#
          addi     r28,r0,8                # r28 = bit pattern = 8
          stb      r28,4(r31)              # Dismiss interrupt
          addi     r31,r0,0                # r31 = 0 = no ISP Handler
          blr
```

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`
ISP Stem:	`clockstem`
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	Value
Parameter:	`0x80000040`

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 ISP stem `clockstem`. Use the following parameters in your definition of the fast clock ISP.

ISP Type:	Fast Clock Handler
ISP Root:	`clockroot`
Address #1:	`0x80000044`
Value #1:	`0x08`
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank

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 Decrementer Clock Driver

Most members of the PowerPC family include a decrementer register which is automatically decremented at a fixed frequency, usually determined by a subdivision of the bus frequency. If interrupts are enabled (*EE = 1* in the *MSR*), the decrementer unconditionally generates an exception through exception vector *0x0900* whenever bit *0* of the register value changes from *0* to *1*.

The AMX decrementer clock driver is ready for use with any PowerPC equipped with the decrementer exception. **Source code** for this AMX clock driver is generated in the Target Configuration Module produced by the AMX Configuration Builder.

To use the AMX decrementer clock driver, you must define the decrementer exception to be an AMX clock as described in Chapter 4.2. Use the following parameters in your definition of the decrementer exception.

Purpose:	AMX Clock
Downcount value:	32-bit downcount value sets clock frequency
Vector number:	AMX vector number to be assigned to decrementer clock

The **downcount value** is a 32-bit positive value used to define the AMX clock interrupt frequency. For most PowerPCs, the decrementer frequency is 1/4 of the bus frequency. The downcount parameter *DCOUNT* can be computed using the following formula:

```
DCOUNT = p * (f/4)
```

p = required decrementer exception period measured in microseconds.
f = decrementer source frequency expressed in MHz.

Example:

p = 1000 µs (for a 1 KHz decrementer clock)
f = 66 MHz

```
DCOUNT = 1000 * (66/4) = 16500
```

The decrementer frequency can also be dynamically adjusted at run time. To do so, configure the decrementer downcount value to be *0*. Your *main()* program must then install the real decrementer value into *long* variable *cjcfdecrvalue* prior to launching AMX. Thereafter, any change which you make to the value of variable *cjcfdecrvalue* will take effect at the next decrementer exception.

You must also provide the **vector number** in the AMX Vector Table which you wish to assign to the decrementer.

Given this definition, your Target Configuration Module will include a clock ISP root named `chdecclk`, a clock initialization procedure `chclockinit` and a clock shutdown procedure `chclockexit`.

When AMX is launched, a very long downcount value is loaded into the decrementer register to prevent interrupts during the launch. The clock initialization procedure `chclockinit` will then install the pointer to the clock ISP root `chdecclk` into the decrementer's vector in the AMX Vector Table and load the correct downcount value into the decrementer register.

Thereafter, the register decrements at a frequency which is PowerPC implementation specific. When the decrementer exception occurs, the AMX exception handler reloads the decrementer value with a new downcount value and branches through the entry in the AMX Vector Table to the ISP root `chdecclk` which generates an AMX clock tick.

You must 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 begin downcounting 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. This procedure will simply install an `rfi` instruction into the decrementer exception vector to force subsequent decrementer exceptions to be ignored.

MPC505 Decrementer

The MPC505 decrementer frequency is 1/4 of the bus frequency or 1/4 of the frequency established by an external oscillator. The frequency source is determined by the state of the `TBS` bit in the System Clock Control Register (`SCCR`).

The MPC505 decrementer must be enabled by setting the `DCE` bit in the `SCCR` and the `PTE` bit in the Periodic Interrupt Control and Select Register (`PICSR`). If these bits are not both set, the MPC505 decrementer will remain disabled.

You must configure the `SCCR` and `PICSR` registers as required by your application prior to launching AMX.

MPC860 Decrementer

The MPC860 decrementer frequency can be set to 1/4 or 1/16 of the bus frequency. The MPC860 decrementer must be enabled by setting the `TBE` bit and clearing the `TBF` bit in the Time Base Control and Status Register (`TBSCR`). If the `TBE` bit is clear, the MPC860 decrementer will remain disabled. If the `TBF` bit is set, the MPC860 decrementer will remain frozen.

You must configure the `TBSCR` register as required by your application prior to launching AMX.

MPC8560 Decrementer

The MPC8560 decrementer frequency can be set to 1/8 of the bus frequency or the frequency of the external *RTC* signal. The MPC8560 time base module must be enabled by setting the *TBEN* bit and selecting the appropriate value for the *SEL_TBCLK* bit in the Hardware Implementation Dependent Register 0 (*HID0*). If the *TBEN* bit is clear, the MPC8560 decrementer will remain disabled.

You must configure the *HID0* register as required by your application prior to launching AMX.

PPC440 Decrementer

The PPC440GP decrementer frequency can be set to the core clock frequency or the frequency of the external *TMR_CLK* signal. The clock source is determined by the state of the *CETE* bit in the Chip Control Register 0 (*CPC0_CR0*).

You must configure the *CPC0_CR0* register as required by your application prior to launching AMX.

5.3.2 8254 Clock Driver

The AMX clock driver for the Intel 8254 counter/timer chip is ready for use on the Motorola Ultra 603 motherboard 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 *CH8254T.C*.

Board support module *ULT603.S*, or your equivalent, must be used to configure the onboard Intel 8259 Interrupt Controller to allow the 8254 timer to generate an *IRQ0* interrupt which is assigned interrupt identification number 0 which maps to AMX vector number 0. To use this module on the Motorola Ultra 603 motherboard platform, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*16*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector the sixteen Ultra 603 device interrupts through the block of 16 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *ULT603.S* is used to identify the interrupt source.

Note that the standard Interrupt Identification Procedure *ch500vd* in board support module *ULT603.S* assigns interrupt identification number 0 to 8254 timer 0. Since the 8254 timer is configured to interrupt via the *IRQ0* interrupt exception, the interrupt identification number maps to vector number 0 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 0).

You must compile clock source module *CH8254T.C*, assemble board support source module *ULT603.S* and link the resulting object modules with the rest of your AMX application.

To use the AMX 8254 clock driver, you must create a fast clock ISP root as described in Chapter 4.4. Use the following parameters in your definition of the fast clock ISP. Note that the interrupt from timer 0 is cleared by writing one byte of value *0x20* to the 8259 Interrupt Controller at device address *0x80000020*.

ISP Type:	Fast Clock Handler
ISP Root:	*ch8254clk*
Address #1:	*0x80000020*
Value #1:	*0x20*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch8254clk* which dismisses the timer interrupt. The timer automatically restarts with the required period.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch8254clk* into the AMX Vector Table at vector number *0*.

The board support module *ULT603.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 *CH8254T.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 8254 Clock Driver

If you wish to use a different 8254 timer channel, change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH8254T.C* and recompile the module. Edit instructions are included in the file.

If you wish to use a different 8254 timer channel, assign a different clock interrupt identification number or AMX vector number or port the 8254 clock driver to a different hardware platform, you will also have to edit and assemble the board support module *ULT603.S*.

The board support module *ULT603.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.3 PPC403 PIT Clock Driver

The AMX clock driver for the IBM PPC403 internal Programmable Interval Timer (PIT) is ready for use on the IBM 403 EVB Evaluation Board. It is configured to use the timer operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH403T.C*.

Board support module *CH403EVB.S*, or your equivalent, must be used to configure the PPC403 PIT timer for use in your application. To use this module on the IBM 403 EVB Evaluation Board, you must define the PPC403 PIT exception vector *0x1000* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	Leave blank
Number of devices:	*1*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the PPC403 exception vector *0x1000* is to be used to vector the internal PIT interrupt through vector number 0 in the AMX Vector Table. There is no need for an Interrupt Identification Procedure.

You must compile clock source module *CH403T.C*, assemble board support source module *CH403EVB.S* and link the resulting object modules with the rest of your AMX application.

To use the PPC403 PIT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch403clk*
ISP Stem:	*ch403is*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch403clk*. The clock ISP stem *ch403is*, located in the board support module *CH403EVB.S*, dismisses the PIT interrupt and restarts the timer with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch403clk* into the AMX Vector Table at vector number *0*.

The board support module *CH403EVB.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 *CH403T.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 PPC403 PIT Clock Driver

If you wish to change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH403T.C* and recompile the module. Edit instructions are included in the file.

If you port the PPC403 PIT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *CH403EVB.S*.

The board support module *CH403EVB.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.4 MPC860 PIT Clock Driver

The AMX clock driver for the Motorola MPC860 internal Programmable Interval Timer (PIT) is ready for use on the Motorola MPC860ADS board. It is configured to use the timer operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH860T.C*.

Board support module *CH860ADS.S*, or your equivalent, must be used to configure the MPC860 PIT timer for use in your application. To use this module on the Motorola MPC860ADS board, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*16*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector the sixteen MPC860 device interrupts through the block of 16 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *CH860ADS.S* is used to identify the interrupt source.

Note that the standard Interrupt Identification Procedure *ch500vd* in board support module *CH860ADS.S* reads the MPC860 interrupt controller and derives interrupt identification number 11 for the PIT timer. The interrupt identification number maps to vector number 11 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 11).

You must compile clock source module *CH860T.C*, assemble board support source module *CH860ADS.S* and link the resulting object modules with the rest of your AMX application.

> Note
>
> MPC860 PIT clock drivers are also provided for use on the Motorola MPC860FADS board and the Motorola MBX860 board. Pick the clock driver module *CH860T.C* from the board specific directory in the AMX *TOOLxx\SAMPLE* directory. Use board support module *FADS860.S* for the Motorola MPC860FADS board. Use board support module *CH860MBX.S* for the Motorola MBX860 board.

To use the MPC860 PIT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch860clk*
ISP Stem:	*ch860clkstem*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch860clk*. The clock ISP stem *ch860clkstem*, located in the board support module *CH860ADS.S*, dismisses the PIT interrupt and restarts the timer with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch860clk* into the AMX Vector Table at vector number *11*.

The board support module *CH860ADS.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 *CH860T.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 MPC860 PIT Clock Driver

If you wish to change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH860T.C* and recompile the module. Edit instructions are included in the file.

If you port the MPC860 PIT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *CH860ADS.S*.

The board support module *CH860ADS.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.5 MPC8260 PIT Clock Driver

The AMX clock driver for the Motorola MPC8260 internal Programmable Interval Timer (PIT) is ready for use on the EST SBC8260 Single Board Computer. It is configured to use the timer operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH8260T.C*.

Board support module *SBC8260.S*, or your equivalent, must be used to configure the MPC8260 PIT timer for use in your application. To use this module on the EST SBC8260 board, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*64*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector the 64 MPC8260 device interrupts through the block of 64 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *SBC8260.S* is used to identify the interrupt source.

Note that the standard Interrupt Identification Procedure *ch500vd* in board support module *SBC8260.S* reads the MPC8260 interrupt controller and derives interrupt identification number 17 for the PIT timer. The interrupt identification number maps to vector number 17 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 17).

You must compile clock source module *CH8260T.C*, assemble board support source module *SBC8260.S* and link the resulting object modules with the rest of your AMX application.

To use the MPC8260 PIT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch8260clk*
ISP Stem:	*ch8260clkstem*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch8260clk*. The clock ISP stem *ch8260clkstem*, located in the board support module *SBC8260.S*, dismisses the PIT interrupt and restarts the timer with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch8260clk* into the AMX Vector Table at vector number *17*.

The board support module *SBC8260.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 *CH8260T.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 MPC8260 PIT Clock Driver

If you wish to change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH8260T.C* and recompile the module. Edit instructions are included in the file.

If you port the MPC8260 PIT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *SBC8260.S*.

The board support module *SBC8260.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.6 MPC823 PIT Clock Driver

The AMX clock driver for the Motorola MPC823 internal Programmable Interval Timer (PIT) is ready for use on the RPX Lite MPC823 board. It is configured to use the timer operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *CH823T.C*.

Board support module *CH823RPX.S*, or your equivalent, must be used to configure the MPC823 PIT timer for use in your application. To use this module on the RPX Lite MPC823 board, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*16*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector the sixteen MPC823 device interrupts through the block of 16 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *CH823RPX.S* is used to identify the interrupt source.

Note that the standard Interrupt Identification Procedure *ch500vd* in board support module *CH823RPX.S* reads the MPC823 interrupt controller and derives interrupt identification number 11 for the PIT timer. The interrupt identification number maps to vector number 11 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 11).

You must compile clock source module *CH823T.C*, assemble board support source module *CH823RPX.S* and link the resulting object modules with the rest of your AMX application.

To use the MPC823 PIT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch823clk*
ISP Stem:	*ch823clkstem*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch823clk*. The clock ISP stem *ch823clkstem*, located in the board support module *CH823RPX.S*, dismisses the PIT interrupt and restarts the timer with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch823clk* into the AMX Vector Table at vector number *11*.

The board support module *CH823RPX.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 *CH823T.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 MPC823 PIT Clock Driver

If you wish to change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH823T.C* and recompile the module. Edit instructions are included in the file.

If you port the MPC823 PIT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *CH823RPX.S*.

The board support module *CH823RPX.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.7 PPC405 PIT Clock Driver

The AMX clock driver for the IBM PPC405 internal Programmable Interval Timer (PIT) is ready for use on the IBM PPC405GP Reference Board. It is configured to use the timer operating at approximately 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file `CH405T.C`.

Board support module `CH405EVB.S`, or your equivalent, must be used to configure the PPC405 PIT timer for use in your application. To use this module on the IBM PPC405GP Reference Board, you must define the external interrupt exception vector `0x0500` as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	`ch500vd`
Number of devices:	`32`
First vector number:	`0`
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector `0x0500` is to be used to vector the thirty two PPC405GP device interrupts through the block of 32 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure `ch500vd` in board support module `CH405EVB.S` is used to identify the interrupt source.

You must also define the PPC405 PIT exception vector `0x1000` as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	Leave blank
Number of devices:	`1`
First vector number:	`32`
Run-time vector checking:	Unchecked

This definition declares that the PPC405GP exception vector `0x1000` is to be used to vector the internal PIT interrupt through vector number 32 in the AMX Vector Table. There is no need for an Interrupt Identification Procedure.

You must compile clock source module `CH405T.C`, assemble board support source module `CH405EVB.S` and link the resulting object modules with the rest of your AMX application.

To use the PPC405 PIT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch405clk*
ISP Stem:	*ch405is*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch405clk*. The clock ISP stem *ch405is*, located in the board support module *CH405EVB.S*, dismisses the PIT interrupt and restarts the timer with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch405clk* into the AMX Vector Table at vector number *32*.

The board support module *CH405EVB.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 *CH405T.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 PPC405 PIT Clock Driver

If you wish to change the timer frequency or use a different AMX vector number, you must edit the definitions in source file *CH405T.C* and recompile the module. Edit instructions are included in the file.

If you port the PPC405 PIT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *CH405EVB.S*.

The board support module *CH405EVB.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.8 MPC5200 GPT Clock Driver

The AMX clock driver for the Motorola MPC5200 internal General Purpose Timer (GPT) is ready for use on the Lite5200 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 *CH5200T.C*.

Board support module *LITE5200.S*, or your equivalent, must be used to configure the MPC5200 GPT timer for use in your application. To use this module on the Lite5200 Evaluation Board, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*84*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector eighty four MPC5200 device interrupts through the block of 84 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *LITE5200.S* is used to identify the interrupt source.

Note that the standard Interrupt Identification Procedure *ch500vd* in board support module *LITE5200.S* reads the MPC5200 interrupt controller and derives interrupt identification number 13 for GPT timer 0. The interrupt identification number maps to vector number 13 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 13).

You must compile clock source module *CH5200T.C*, assemble board support source module *LITE5200.S* and link the resulting object modules with the rest of your AMX application.

To use the MPC5200 GPT clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch5200clk*
ISP Stem:	*ch5200clkstem*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch5200clk*. The clock ISP stem *ch5200clkstem*, located in the board support module *LITE5200.S*, dismisses the GPT interrupt. The timer automatically restarts with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch5200clk* into the AMX Vector Table at vector number *13*.

The board support module *LITE5200.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 *CH5200T.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 MPC5200 GPT Clock Driver

If you wish to change the timer frequency, use a different GPT timer or use a different AMX vector number, you must edit the definitions in source file *CH5200T.C* and recompile the module. Edit instructions are included in the file.

If you port the MPC5200 GPT clock driver to a different hardware platform, you will also have to edit and assemble the board support module *LITE5200.S*.

The board support module *LITE5200.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

5.3.9 MPC8560 Global Timer Clock Driver

The AMX clock driver for the Motorola MPC8560 internal global timer is ready for use on the MPC8560ADS 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 *CH8560T.C*.

Board support module *ADS8560.S*, or your equivalent, must be used to configure the MPC8560 global timer for use in your application. To use this module on the MPC8560ADS board, you must define the external interrupt exception vector *0x0500* as described in Chapter 4.2 using the following parameters.

Purpose:	Interrupt
ID procedure:	*ch500vd*
Number of devices:	*80*
First vector number:	*0*
Run-time vector checking:	Unchecked

This definition declares that the external interrupt exception vector *0x0500* is to be used to vector eighty MPC8560 device interrupts through the block of 80 vectors in the AMX Vector Table beginning at vector number 0. The Interrupt Identification Procedure *ch500vd* in board support module *ADS8560.S* is used to identify the interrupt source.

Note that the MPC8560 clock initialization function *chclockinit* programs the MPC8560 interrupt controller so that the standard Interrupt Identification Procedure *ch500vd* in board support module *ADS8560.S* derives interrupt identification number 2 for global timer 0. The interrupt identification number maps to vector number 2 in the AMX Vector Table (base vector 0 plus timer interrupt identification number 2).

You must compile clock source module *CH8560T.C*, assemble board support source module *ADS8560.S* and link the resulting object modules with the rest of your AMX application.

To use the MPC8560 global timer clock driver, you must create a clock ISP root as described in Chapter 4.3. Use the following parameters in your definition of the clock ISP.

ISP Type:	Clock Handler
ISP Root:	*ch8560clk*
ISP Stem:	*ch8560clkstem*
Stem Language:	Assembly
ISP Handler:	leave blank
Parameter Type:	(none)
Parameter:	leave blank

The resulting Target Configuration Module will include a clock ISP root named *ch8560clk*. The clock ISP stem *ch8560clkstem*, located in the board support module *ADS8560.S*, dismisses the global timer interrupt. The timer automatically restarts with the required period. There is no need for an ISP Handler.

The clock initialization procedure *chclockinit* will install the pointer to the clock ISP root *ch8560clk* into the AMX Vector Table at vector number *2*.

The board support module *ADS8560.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 *CH8560T.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 MPC8560 Global Timer Clock Driver

If you wish to change the timer frequency, use a different global timer or use a different AMX vector number, you must edit the definitions in source file *CH8560T.C* and recompile the module. Edit instructions are included in the file.

If you port the MPC8560 global timer clock driver to a different hardware platform, you will also have to edit and assemble the board support module *ADS8560.S*.

The board support module *ADS8560.S* also includes a board initialization procedure *chbrdinit* which must be modified to initialize the interrupt system for your particular board. Procedure *chbrdinit* must be called from your *main* program prior to launching AMX.

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
...HDW             PROC,CACHE
...CACHE           FNCACHE,ICSIZE,ICPARAM,DCSIZE,DCPARAM
...DELAY           CPUFREQ
...IRQ             NVEC,NBLOCK
...MSR             MSRMASK
...EVTABLE         EVBASE,EVBASE2
...EVFATAL         EVMASK,EVOFS
;
;           Interrupt exception definitions (one line per exception)
...EHINT           VOFS,VIDPROC,VNBASE,VNCOUNT,VNCHECK
;
;           User exception service procedure definitions
;           (one line for each definition)
...EHUSERA         VOFS,EHPROC,EHPARAM
...EHUSERC         VOFS,EHPROC,EHPARAM
...EHUSERF         VOFS,EHPROC,EHPARAM
;
;           ISP definitions (one line for each ISP)
...ISPA            ISPROOT,STEM,HANDLER,VNUM,PARAM,PARTYPE
...ISPC            ISPROOT,STEM,HANDLER,VNUM,PARAM,PARTYPE
;
;           Conforming decrementer clock ISP (no user code required)
...CLKDECR         DCOUNT
;           Conforming fast clock ISP (no user code required)
...CLKFAST         CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
;           or conforming clock ISP (coded in assembly language)
...CLKA            CLKROOT,CLKSTEM,CLKHAND,VNUM,PARAM,PARTYPE
;           or conforming clock ISP (coded in C)
...CLKC            CLKROOT,CLKSTEM,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
;
;           Null Functions (optional; one line for each null function)
...NULLFN          FNNAME
```

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 PPC32 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*
PERM	0 if the AMX launch is temporary 1 if the AMX launch is permanent

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

...HDW	*PROC,CACHE*
PROC	Processor identifier
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. *PROC* must be one of:

401, 401A1, 401X2, 401B3, 401GF,	{IBM PPC401 or equivalent}
403, 403GA, 403GC,	{IBM PPC403 or equivalent}
405, 405B3, 405GP,	{IBM PPC405 or equivalent}
440,	{IBM PPC440 or equivalent}
505, 509, 555,	{Motorola MPC5xx or equivalent}
5553,	{Freescale MPC555x or equivalent}
601, 602, 603, 604,	{Motorola MPC60x or equivalent}
603e, 5200, 8240, 8260, *8280, 8349,*	{Motorola MPC603e or equivalent}
740, 750, 7400,	{Motorola MPC7xx or equivalent}
801, 821, 823, 850, 855, 860,	{Motorola MPC8xx or equivalent}
8560	{Motorola MPC85xx or equivalent}

The *CACHE* parameter can be used to instruct AMX to enable the PowerPC instruction and data caches when AMX is launched. If the processor 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	PowerPC 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.

Device Interrupt Support

The Target Parameter File includes a definition of the interrupt support which AMX must provide.

```
...IRQ              NVEC,NBLOCK
```

NVEC	Number of entries in the AMX Vector Table
NBLOCK	Number of AMX Interrupt Request Blocks

AMX provides an AMX Vector Table through which it dispatches all device interrupt service requests. The parameter *NVEC* defines the number of entries (vectors) in the AMX Vector Table which you require for your application. This number can be determined by observing the highest AMX vector number which you allocate to any interrupt exception using the *...EHINT* directive.

AMX uses an Interrupt Request Block to defer execution of any ISP Handler while AMX is in any critical region of code. The parameter *NBLOCK* defines the number of Interrupt Request Blocks needed to accommodate the worst case interrupt deferral which may occur in your application.

Exception Handler MSR Adjustment

The Target Parameter File includes a machine state definition common to all AMX exception handlers.

```
...MSR              MSRMASK
```

MSRMASK	AMX exception handler *MSR* update mask

When an exception occurs, the PowerPC copies the current machine state register (*MSR*) to save/restore register *SRR1* and then updates the *MSR* in an exception dependent fashion. Because of this update, some of the configuration parameters in the *MSR* may actually be altered leaving memory or devices unaccessible by the exception handler.

To overcome this constraint, the *...MSR* directive can be used to define which, if any, of the control bits in the *MSR* are to be restored by the AMX exception handler to their state prior to the exception. The *MSR* control bits are identified by setting the corresponding bits in parameter *MSRMASK*.

If you do not need to restore the state of any *MSR* control bits within AMX exception handlers, set *MSRMASK* to *0* or omit the *...MSR* directive.

Exception Vector Base Address

The Target Parameter File includes a definition which permits AMX to derive or set the base address of the Exception Vector Table.

```
...EVTABLE          EVBASE
```

EVBASE = *0xVVVVVVVV* = A(Exception Vector Table)

The *EVBASE* parameter is used to specify the base address in memory at which the Exception Vector Table is located.

If parameter *EVBASE* is set to *-1*, AMX will derive the base address of the processor's Exception Vector Table at launch time according to the processor type specified by parameter *PROC* in the *...HDW* directive.

For processors such as the **MPC5xx, MPC5200, MPC601, MPC603, MPC604, MPC7xx, MPC8xx and MPC82xx** which adhere to the PowerPC Architecture specification, AMX uses the *IP* bit in the machine state register (*MSR*) to determine the address of the Exception Vector Table. If *IP = 0,* the table is at address *0*; otherwise the table is at address *0xFFF00000*. If parameter *EVBASE* is set to *-1*, AMX will read the state of the *IP* bit and determine the base address accordingly. If *EVBASE* is *0* or *1*, AMX will set *IP* equal to *EVBASE* thereby adjusting the base address to *0* or *0xFFF00000*.

For the **PPC40x** families, the Exception Vector Prefix Register (*EVPR*) determines the base address of the Exception Vector Table. If parameter *EVBASE* is set to *-1*, AMX will read the *EVPR* to determine the base address. If *EVBASE* is any other value, AMX will copy *EVBASE* into the *EVPR*, thereby establishing *EVBASE* as the base address. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

For processors based on the PowerPC Book E specification, such as the **MPC555x, MPC85xx and PPC440** families, the Interrupt Vector Prefix Register (*IVPR*) determines the base address of the Exception Vector Table. If parameter *EVBASE* is set to *-1*, AMX will read the *IVPR* to determine the base address. If *EVBASE* is any other value, AMX will copy *EVBASE* into the *IVPR*, thereby establishing *EVBASE* as the base address. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

For the **MPC602**, the *IP* bit in the machine state register (*MSR*) and the Interrupt Base Register (*IBR*) determine the base address of the Exception Vector Table. If *IP = 1*, the table is at address *0xFFF00000*. If *IP = 0,* critical exception vectors are unconditionally at address *0* and all others are at the base address determine by the content of the Interrupt Base Register (*IBR*). If parameter *EVBASE* is set to *-1*, AMX will read the state of the *IP* bit and the content of the *IBR*, if necessary, to derive the base address. If parameter *EVBASE* is set to *1*, AMX will set the *IP* bit to *1* thereby forcing the base address to be *0xFFF00000*. If *EVBASE* is any other value, AMX will set the *IP* bit to *0* and copy *EVBASE* into the *IBR*, thereby establishing *EVBASE* as the base address for all but critical exceptions. Note that a valid base address must have the least significant 16 bits of the address set to *0*.

AMX requires that the Exception Vector Table be located in RAM. If your Exception Vector Table must be in ROM, use the following alternate form of the `...EVTABLE` directive to force AMX to use a shadowed Exception Vector Table in RAM at memory address `EVBASE2` as described in Chapter 3.7. Note that a valid base address must have the least significant 8 bits of the address set to `0`.

```
...EVTABLE        -1,EVBASE2
```

`EVBASE2` = `0xVVVVVVVV` = A(Pseudo Exception Vector Table)

Fatal Exception Vector Definitions

You can instruct AMX to treat any of the PowerPC exceptions as fatal. When so instructed, AMX will create a special exception handler and install it in the particular entry in the Exception Vector Table. When the exception occurs, the AMX exception handler will call the Fatal Exception Procedure `cjksfatalexh` in module `CJ382UF.C` as described in Chapter 3.1.

To declare exception vectors as fatal, insert one or more `...EVFATAL` directives into your Target Parameter File. The allowable forms of the directive are as follows.

```
...EVFATAL        EVMASK
```
or
```
...EVFATAL        EVMASK2,0x2000
```
or
```
...EVFATAL        1,EVOFS
```

`EVMASK` = `0xMMMMMMMM` = AMX exception vector mask
`EVMASK2` = `0xMMMMMMMM` = Extended exception vector mask
`EVOFS` = `0xVVVV` = vector offset in the Exception Vector Table

Exception vectors are located at intervals of `0x0100` throughout the Exception Vector Table. Parameter `EVMASK` is used to identify each of the 32 exception vectors from vector offset `0x0000` to `0x1F00`. The least significant bit of `EVMASK` corresponds to vector offset `0x0000`. The most significant bit of `EVMASK` corresponds to vector offset `0x1F00`.

Set bit `N` of the `EVMASK` exception vector bit mask for each of the exceptions which are to be treated as fatal by AMX. For example, set this parameter to `0x001F11DE` to allow AMX to treat all of the Motorola MPC603 exceptions as fatal except the external interrupt (vector offset `0x0500`) and decrementer (vector offset `0x0900`) exceptions. Bits in the exception vector mask are defined in Table 3.1-1.

Some PowerPC implementations (such as the Motorola MPC601 and the IBM PPC403) provide additional exception vectors at vector offset `0x2000` and above. Although AMX does not define exception vector masks for these exceptions, the second form of the `...EVFATAL` directive can still be used to declare any of the exceptions from `0x2000` to `0x3F00` as fatal. In this case, the least significant bit of `EVMASK2` corresponds to vector offset `0x2000`. The most significant bit of `EVMASK2` corresponds to vector offset `0x3F00`.

The IBM PPC403, PPC405, PPC440, Freescale MPC555x and Motorola MPC85xx implement three short exception vectors at vector offsets *0x1000*, *0x1010* and *0x1020*. A subset of these processors also add two short exception vectors at vector offsets *0x0F00*, *0x0F20*. To declare these exceptions as fatal, you must use the third form of the *...EVFATAL* directive. A separate directive is required for each of the three exceptions. The mask must be *1* and the *EVOFS* parameter is the vector offset for the short exception vector being declared fatal.

> Note
>
> When using AMX with a debugger, do not declare any of the exceptions which the debugger services to be fatal.

Interrupt Exception Handlers

The Target Parameter File defines the AMX interrupt exception handlers which are needed to dispatch all device interrupt service requests. A separate definition is required for each PowerPC interrupt exception which AMX must service. Each definition overrides the fatal AMX handler, if any, which may have been specified in the *...EVFATAL* directive.

If **multiple devices** generate interrupts through a single exception vector, the interrupt exception handler definition is as follows.

```
...EHINT          VOFS,VIDPROC,VNBASE,VNCOUNT,VNCHECK
```

VOFS	Vector offset in the Exception Vector Table
VIDPROC	Name of the Interrupt Identification Procedure
VNBASE	Base vector number in the AMX Vector Table
VNCOUNT	Number of vectors, beginning at *VNBASE*, required by devices interrupting through the exception vector at vector offset *VOFS*
VNCHECK	*0* if run-time vector number checking is disabled *1* if run-time vector number checking is enabled

The vector offset *VOFS* is the displacement in the PowerPC Exception Vector Table at which the processor begins execution when the interrupt exception occurs.

If more than one device can cause the interrupt exception to occur, you must provide an Interrupt Identification Procedure (see Chapter 3.2). Parameter *VIDPROC* is the name of that procedure. 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 through an AMX interrupt exception. Parameter *VNBASE* defines the base vector number assigned by you to the devices attached to the particular interrupt exception. The interrupt identification number provided by your Interrupt Identification Procedure is added to *VNBASE* to derive the AMX vector number for the device.

Parameter *VNCOUNT* defines the number of AMX vectors starting at vector number *VNBASE* which are needed to meet the needs of the devices attached to the particular interrupt exception. The vector numbers *VNBASE* to *VNBASE+VNCOUNT-1* must be within the range of AMX vectors allocated by parameter *NVEC* in the *...IRQ* directive. For example, if 16 devices are assigned to the same interrupt exception but are identified with integers from 8 to 15 and 56 to 63, you could use a *VNBASE* of *0* and *VNCOUNT* of *64* to easily map the interrupt identifiers one-to-one with their vector numbers.

The AMX interrupt exception handler can check that the derived vector number lies within the range *VNBASE* to *VNBASE+VNCOUNT-1* in the AMX Vector Table. If it does not, AMX calls the Fatal Exception Handler indicating that an unidentified interrupt exception (exception variety *CJ_PRXVUIR*) occurred via vector offset *VOFS*.

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

If a **single device** generates an interrupt through a dedicated exception vector, the interrupt exception handler definition is as follows.

```
...EHINT            VOFS,,VNBASE,1,0
```

VOFS	Vector offset in the Exception Vector Table
VNBASE	Vector number in the AMX Vector Table

This form of the *...EHINT* directive is a special case of the general multiplexed form. In this case, parameter *VIDPROC* is omitted, *VNCOUNT* is *1* and *VNCHECK* is *0*.

The vector offset *VOFS* is the displacement in the PowerPC Exception Vector Table at which the processor begins execution when the dedicated device interrupt exception occurs.

An entry in the AMX Vector Table is reserved for every device serviced through an AMX interrupt exception. Parameter *VNBASE* defines the vector number assigned by you to the device attached to the particular dedicated interrupt exception. The vector number *VNBASE* must be within the range of AMX vectors allocated by parameter *NVEC* in the *...IRQ* directive. No run-time vector number checking is performed by the AMX interrupt exception handler.

User Exception Service Procedures

The Target Parameter File can be used to provide access to your own custom user exception service procedure without having to create your own exception handler and install it in the Exception Vector Table. Each such user defined exception overrides the fatal AMX handler, if any, which may have been specified in the *...EVFATAL* directive.

Each custom user exception definition assumes one of the following forms.

Use directive *...EHUSERA* if your user exception service procedure is coded in assembly language.

Use directive *...EHUSERC* if your user exception service procedure is coded in C.

Use directive *...EHUSERF* if your user exception service procedure is coded in C and requires access to an AMX register structure *cjxregs* which completely defines the machine state at the time the exception occurred.

```
...EHUSERA          VOFS,EHPROC,EHPARAM
...EHUSERC          VOFS,EHPROC,EHPARAM
...EHUSERF          VOFS,EHPROC,EHPARAM
```

VOFS	Vector offset in the Exception Vector Table
EHPROC	Name of your user exception service procedure
EHPARAM	Integer parameter passed to procedure *EHPROC*

The vector offset *VOFS* is the displacement in the PowerPC Exception Vector Table at which the processor begins execution when the particular exception occurs.

EHPROC is the name of your public assembly language or C procedure which AMX will call whenever the exception is generated. The procedure must abide by the calling conventions described in Chapter 3.1.

EHPARAM is an optional integer parameter in the range *0* to *0x000FFFFF* which will be merged with the exception variety code *CJ_PRXVUSR* and passed to your user exception service procedure. If no parameter is required, omit *EHPARAM* from the directive. In this case, the parameter received by your procedure will simply be the exception variety code *CJ_PRXVUSR* merged with the vector offset *VOFS*.

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 ISP stem is coded in C or *...ISPA* if its ISP stem is coded in assembly language.

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

ISPROOT	Name of the ISP root entry point
STEM	Name of the public device ISP stem
HANDLER	Name of the public device ISP Handler
VNUM	AMX vector number assigned to the device
PARAM	Parameter for use by ISP stem and ISP Handler
PARTYPE	Parameter *PARAM* type

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

If your ISP stem and ISP Handler require 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 ISP stem and ISP Handler require 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. *VNUM* must be a value from *VNBASE* to *VNBASE+VNCOUNT-1* (see directive *...EHINT*), the range of AMX vectors assigned to the exception through which the device generates interrupts.

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.

Nonconforming ISP Declarations

The Target Parameter File must include a definition of an ISP root for each nonconforming 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 ISP stem is coded in C or *...ISPA* if its ISP stem is coded in assembly language.

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

ISPROOT	Name of the ISP root entry point
STEM	Name of the public device ISP stem There can be no ISP Handler
VNUM	AMX vector number assigned to the device
PARAM	Parameter for use by ISP stem
PARTYPE	Parameter *PARAM* type

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

If your ISP stem 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 ISP stem 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. *VNUM* must be a value from *VNBASE* to *VNBASE+VNCOUNT-1* (see directive *...EHINT*), the range of AMX vectors assigned to the exception through which the device generates interrupts.

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 ISP stem is coded in C or *...CLKA* if its ISP stem is coded in assembly language. The clock ISP root can be declared using *...CLKFAST* if an ISP Handler is not required to service the clock.

```
...CLKC         CLKROOT,CLKSTEM,CLKHAND,VNUM,PARAM,PARTYPE
...CLKA         CLKROOT,CLKSTEM,CLKHAND,VNUM,PARAM,PARTYPE
...CLKFAST      CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
```

CLKROOT	Name of the clock ISP root entry point
CLKSTEM	Name of the public clock device ISP stem
CLKHAND	Name of the public clock device ISP Handler
VNUM	AMX vector number assigned to the clock device
PARAM	Parameter for use by ISP stem and ISP Handler
PARTYPE	Parameter *PARAM* type

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

If your clock ISP stem and ISP Handler require 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 ISP stem and ISP Handler require 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 bytes 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*.

Decrementer Exception

Most members of the PowerPC family include a decrementer register which is automatically decremented at a fixed frequency, usually determined by a subdivision of the bus frequency. If interrupts are enabled (*EE = 1* in the *MSR*), the decrementer unconditionally generates an exception through exception vector *0x0900* whenever the most significant bit (bit *0*) of the register value changes from *0* to *1*.

Decrementer as an AMX Clock

The decrementer can be used as the source of AMX clock ticks. To do so, you must treat the decrementer exception as a clock interrupt by including the following two directives in your Target Parameter File.

```
...EHINT          0x0900,,VNBASE,1,0
...CLKDECR        DCOUNT
```

VNBASE is the vector number in the AMX Vector Table through which the decrementer interrupt will be vectored. *DCOUNT* is the 32-bit positive value used to define the AMX clock interrupt frequency (See Chapter 5.3.1).

Decrementer Interrupts

You can use the decrementer exception as a special source of AMX interrupts. To treat the decrementer exception as a single device interrupt assigned to AMX vector number *VNBASE*, include the following directive in your Target Parameter File.

```
...EHINT          0x0900,,VNBASE,1,0
```

To treat the decrementer exception as a multiplexed device interrupt assigned to the set of *VNCOUNT* vectors beginning at AMX vector number *VNBASE*, include the following directive in your Target Parameter File.

```
...EHINT          0x0900,VIDPROC,VNBASE,VNCOUNT,VNCHECK
```

The Interrupt Identification Procedure *VIDPROC* can be used to restart the decrementer by loading a new value into the decrementer register.

You must define a conforming AMX Interrupt Service Procedure (ISP) for each of the artificial device (or devices) which your decrementer represents. Each ISP is defined using a *...ISPA* or *...ISPC* directive.

Custom Decrementer Exception

You can use the decrementer exception for you own purposes. To install an AMX exception handler which will call your custom user exception service procedure, include the following directive in your Target Parameter File. *...EHUSERx* is one of the directives *...EHUSERA*, *...EHUSERC* or *...EHUSERF*. Your user exception service procedure *EHPROC* can be used to restart the decrementer by loading a new value, possibly your parameter *EHPARAM*, into the decrementer register.

```
...EHUSERx          0x0900,EHPROC,EHPARAM
```

Ignoring the Decrementer Exception

Most PowerPC implementations do not permit the decrementer exception to be inhibited. Consequently, an exception handler for the decrementer exception must be provided.

When your Target Parameter File is used to create your Target Configuration Module, any *...EHUSERx* or *...EHINT* reference to the decrementer exception vector *0x0900* is noted. If no reference is detected, the decrementer exception vector *0x0900* is plugged by AMX with an *rfi* instruction to ignore the exception which, although unused, cannot be inhibited.

In some cases, you may wish to allow a decrementer exception handler which is in place prior to launching AMX to retain control of the decrementer exception. If you want AMX to leave the decrementer exception untouched, insert the following *...EHINT* directive in your Target Parameter File.

```
...EHINT            0x0900,,-1,0,0
```

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.

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.

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 PPC32 Target Configuration Template File *CJ382HDW.CT* and a Target Parameter File called *HDWCFG.UP* is as follows:

```
CJ382CG HDWCFG.UP CJ382HDW.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 PPC32 Service Procedures

B.1 Summary of Services

AMX PPC32 provides a collection of target dependent AMX service procedures for use with the PowerPC 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 *MSR*
cjcfei	Enable interrupts
cjcfflagrd	Read the machine state register (*MSR*)
cjcfflagwr	Write to the machine state register (*MSR*)
cjcfhwbatrd	Read the *IBATn* and *DBATn* registers
cjcfhwbatwr	Write to the *IBATn* and *DBATn* registers
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
cjcfmodmsr	Read/modify the machine state register (*MSR*)
cjcfmodhid0	Read/modify hardware implementation register *0* (*HID0*)
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

Processor and C Interface Procedures (class *cf*) (cont'd)

cjcfstkjmp	Switch stacks and jump to a new procedure
cjcftag	Convert a string to an AMX tag value
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* or in your Target Configuration Module. Prototypes will be found in file *CJZZZIF.H*. The register array structures *cjxregs* and *cjxfpregs* which they use are defined in file *CJZZZKT.H*.

cjcffpregld	Load PowerPC floating point registers from a register array
cjcffpregst	Store PowerPC floating point registers into a register array
cjcfregld	Load PowerPC general registers from a register array
cjcfregst	Store PowerPC general registers into a register array
cjcfsint	Generate a software initiated exception

B.2 Service Procedures

A description of all processor dependent AMX PPC32 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 ISP 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 state. However, AMX PPC32 does not actually disable interrupts to close the critical sections of code within its procedures. Instead, AMX leaves the interrupt state unaltered but allows only quick interrupts during its critical operations. Refer to Chapter 3 for a detailed descripton of the AMX interaction with the PowerPC™ interrupt system. 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
■	■	□	Untouched during critical sections but enabled upon return
■	□	■	Untouched. If interrupts are enabled upon entry, only quick interrupts will be allowed during critical sections.
■	■	■	Never disabled, possibly briefly enabled and then, prior to return, restored to the state in effect upon entry to the procedure. If interrupts are enabled upon entry, only quick interrupts will be allowed during critical sections.

The warning (Not in ISP) will be present as a reminder that when the ISP 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 ISP 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 (ISP Handler only) ■ 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 machine state register (*MSR*).

The *EE* bit in the machine state register is reset to *0* to disable interrupts or set to *1* to enable interrupts.

Restrictions An ISP stem must not use any of these procedures. An ISP Handler can use all of these procedures.

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

These procedures must NOT be used prior to launching AMX or after exiting from AMX. Therefore, you must NOT call these procedures from your *main()* program before or after your call to *cjkslaunch()*.

See Also *cjcfflagrd, cjcfflagwr, cjcfmodmsr*

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 machine state register (*MSR*).

Cjcfflagwr updates the machine state register (*MSR*) by writing the parameter *flags* directly to the register.

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

Returns *Cjcfflagrd* returns the actual state of the machine state register.
Cjcfflagwr returns nothing.

Restrictions These procedures can be used to manipulate the machine state register (*MSR*) 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()*.

> Warning
>
> Once AMX has been launched, you can use *cjcfflagrd* to read the state of the machine state register, thereby capturing the current machine state. However, once AMX has been launched, you must NOT use *cjcfflagwr* to write the value acquired by *cjcfflagrd* to the *MSR*. Failure to observe this restriction may lead to an AMX malfunction.

Once AMX has been launched, you MUST use procedure *cjcfmodmsr* to alter configuration and control bits in the *MSR*.

See Also *cjcfdi, cjcfdiprev, cjcfei, cjcfmodmsr*

cjcfhwbatrd
cjcfhwbatwr

cjcfhwbatrd
cjcfhwbatwr

Purpose **Read/Write IBATn and DBATn Registers**

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

Setup Prototypes are in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjcfhwbatrd(struct cjxbats *batp);
void CJ_CCPP cjcfhwbatwr(struct cjxbats *batp);
```

Description *batp* is a pointer to a *cjxbats* structure which defines the upper and lower 32-bit values in each of the four or eight *IBATn* and *DBATn* registers of the PowerPC. Structure *cjxbats* is defined in AMX header file *CJ382KS.H*.

Cjcfhwbatrd reads the *IBATn* and *DBATn* registers and transfers the values to the appropriate entries in the *cjxbats* structure at **batp*.

Cjcfhwbatwr fetches the values from the appropriate entries in the *cjxbats* structure at **batp* and writes them to the *IBATn* and *DBATn* registers.

Interrupts □ Disabled □ Enabled □ Restored

Returns Nothing

Cjcfhwbatrd returns the *IBATn* and *DBATn* values in the *cjxbats* structure at **batp*.

Note These procedures can be called even if your Target Parameter File indicates that you are targeting a PowerPC processor with no *xBATn* registers. If you are targeting a PowerPC processor with only four *xBATn* registers, you do not have to provide initial values for the extra four *xBATn* registers in structure *cjxbats*. The procedures only affect the *xBATn* registers which exist.

These procedures can be used 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()*.

Restrictions These procedures do not inhibit interrupts. It is recommended that interrupts be disabled prior to modifying the *xBATn* registers and restored shortly thereafter.

Example The following example illustrates the steps needed to condition the *xBATn* registers for use on the Motorola Ultra 603 motherboard platform. The setup replicates the operations performed in assembly language by procedure *chbrdinit()* in the Ultra 603 board support module *ULT603.S*.

Example (continued)

```
#include "CJZZZ.H"

struct cjxbats batvalues = {
{  { 0x00001FFF,0x00000002 },/* IBAT0 upper & lower       */
   { 0x00000000,0x00000000 },/* IBAT1                     */
   { 0xF0001FFF,0xF0000001 },/* IBAT2                     */
   { 0x00000000,0x00000000 },/* IBAT3                     */
   },
{  { 0x00001FFF,0x00000002 },/* DBAT0 upper & lower       */
   { 0xBF8000FF,0xBF80003A },/* DBAT1                     */
   { 0xF0001FFF,0xF0000001 },/* DBAT2                     */
   { 0x80001FFF,0x8000003A } /* DBAT3                     */
   },
         /* xBAT4-7 do not exist on the MPC603e           */
{  {0,0},{0,0},{0,0},{0,0} },/* IBAT4-7 upper & lower     */
{  {0,0},{0,0},{0,0},{0,0} } /* DBAT4-7 upper & lower     */
   };

#define HIDMASK = (CJ_MAHID0ICE | CJ_MAHID0DCE)


void main(void)
{
   CJ_T32U msr;

   /* Read and save current MSR                            */
   /* Disable interrupts (EE = 0)                          */
   /* Disable instruction/data translation (IR = DR = 0)   */
   /* Disable instruction/data caches (ICE = DCE = 0)      */
   msr = cjcfflagrd();
   cjcfflagwr(msr & ~(CJ_MAMSREE | CJ_MAMSRIR | CJ_MAMSRDR);
   cjcfmodhid0(HIDMASK, 0);

   /* Write IBATn and DBATn registers                      */
   cjcfhwbatwr(&batvalues);

   /* Enable instruction/data caches (ICE = DCE = 1)       */
   /* Enable instruction/data translation (IR = DR = 1)    */
   /* Restore interrupts (EE = previous state)             */
   cjcfmodhid0(HIDMASK, HIDMASK);
   cjcfflagwr(msr | (CJ_MAMSRIR | CJ_MAMSRDR));

   cjkslaunch();              /* Launch AMX                */
   }
```

See Also *cjcfhwbflush, cjcfhwdflush, cjcfhwiflush, cjcfmodhid0*

> Note
>
> If you initialize the *xBATn* registers in your *main()* program, be sure to edit and assemble source module *ULT603.S*, *SBC8260.S* or *LITE5200.S* to avoid conflicts.

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.

cjcfhwbcache cjcfhwdcache cjcfhwicache

cjcfhwbcache cjcfhwdcache cjcfhwicache

Purpose **Flush and Enable/Disable Caches**

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

Setup Prototypes are 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 a PowerPC 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. In particular, PPC403 users MUST adhere to the cache initialization rules defined there.

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

Use caution when calling these procedures or system performance will be degraded, especially if the cache sizes are large. In particular, ISP stems must not use these procedures.

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. The cache control bits in the *MSR*, *HID0*, *IBATn* and *DBATn* registers are not affected.

Note These procedures can be called even if your Target Parameter File indicates that you are targeting a PowerPC 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 *cjcfmodmsr* 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 *MSR* 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. ISP stems must 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 Prototypes in file *CJZZZTF.H* or macros 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 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 *)0x8000002DL)
                                /* Console data register     */
#define CONDATA ((CJ_T8 *)0x8000002FL)


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 Prototypes are 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*

cjcfmcopy cjcfmset

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 Prototypes are 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 Machine State 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 cjcfmodmsr(CJ_T32U mask, CJ_T32U value);
```

Description Use *cjcfmodmsr* to read/modify the machine state register (*MSR*).

mask is a 32-bit mask defining the bits in the machine state register which are to be modified. Set *mask* to *0* to read the machine state 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 *cjcfmodmsr* does not explicitly affect interrupts. However, the processor interrupt state can be altered by *cjcfmodmsr* according to the *EE* bit settings in your *mask* and *value* parameters.

Returns *Cjcfmodmsr* returns the previous state of the machine state register (*MSR*). If *mask* is not *0*, the bits in the *MSR* register specified by *mask* are updated with the values provided in *value*.

Restrictions An ISP stem must not use this procedure. An ISP Handler can use this procedure.

Interrupts are not explicitly affected by this procedure. The final interrupt state will depend on the previous value of the *EE* bit in the *MSR* and the values specified for the *EE* bit by parameters *mask* and *value*.

This procedure must NOT be used prior to launching AMX or after exiting from AMX. Therefore, you must NOT call this procedure from your *main()* program before or after your call to *cjkslaunch()*. Use *cjcfflagrd* and *cjcfflagwr* for this purpose.

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

Purpose **Read or Modify Hardware Implementation Register 0 (*HID0*)**

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

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
CJ_T32U CJ_CCPP cjcfmodhid0(CJ_T32U mask, CJ_T32U value);
```

Description Use *cjcfmodhid0* to read/modify hardware implementation register *0* (*HID0*).

mask is a 32-bit mask defining the bits in hardware implementation register *0* which are to be modified. Set *mask* to *0* to read *HID0* 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 □ Disabled □ Enabled □ Restored

Returns *Cjcfmodhid0* returns the previous state of hardware implementation register *0* (*HID0*). If *mask* is not *0*, the bits in the *HID0* register specified by *mask* are updated with the values provided in *value*.

Note This procedure can be used 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()*.

Restrictions This procedure does not inhibit interrupts. It is recommended that interrupts be disabled prior to modifying the *HID0* register and restored shortly thereafter.

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

cjcfout8 cjcfout16 cjcfout32

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 Prototypes in file *CJZZZTF.H* or macros 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 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 *)0x8000002DL)
                              /* Console data register      */
#define CONDATA ((CJ_T8 *)0x8000002FL)


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 long aligned block of memory for use as a stack.

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

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*

cjcfvol8 cjcfvol16 cjcfvol32 cjcfvolpntr

cjcfvol8 cjcfvol16 cjcfvol32 cjcfvolpntr

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 Prototypes in file *CJZZZTF.H* or macros 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);
   }
```

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 *...EHINT* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table defined by you using the *...IRQ* directive in your Target Parameter File.

oldproc is a pointer to storage for a copy of the ISP root 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 *...EHINT* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table defined by you using the *...IRQ* directive in your Target Parameter File.

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 *...EHINT* directives in your Target Parameter File. Vector numbers range from *0* to *nvec-1* where *nvec* is the size of the AMX Vector Table defined by you using the *...IRQ* directive in your Target Parameter File.

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 PPC32 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.

CJ382ROP.LKT	AMX ROM Option toolset dependent Link Specification Template
CJ382ROP.CT	AMX ROM Option Template
CJ382RAC.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 *CJ382ROP.CT* to produce an assembly language ROM Option Module *CJ382ROP.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:

```
CJ382CG HDWCFG.UP CJ382ROP.CT CJ382ROP.S
```

The ROM Option Module *CJ382ROP.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.

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 *CJ382RAC.CT* to produce an assembly language ROM Access Module *CJ382RAC.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:

```
CJ382CG HDWCFG.UP CJ382RAC.CT CJ382RAC.S
```

The ROM Access Module *CJ382RAC.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 *CJ382RAC.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 *CJ382.A*.

Note that you must still include the AMX Library *CJ382.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 PowerPC™ Architecture includes instruction caches and data caches. However, the cache sizes and control methods are PowerPC implementation dependent. To accommodate the differences, AMX PPC32 provides 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 PPC32 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, do not use the low level cache control functions. Instead, they use the cache flush and invalidate instructions defined by the PowerPC Architecture and common to all PowerPC processors.

> 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.

Cache Initialization

When power is first applied to the PowerPC, 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 PowerPC 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 PowerPC 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 PPC32 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 PowerPC caches. Each module supports a specific cache type. Each cache type is given a name *XXX* identifying a particular PowerPC 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 PowerPC processor. In most cases, the parameters simply accommodate different cache sizes for each particular PowerPC 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.

> 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 assembly language source files.

CH403CAS.S	PPC403 style cache	(*ICCR* and *DCCR* control)
CH405CAS.S	PPC405 style cache	(*ICCR* and *DCCR* control)
CH440CAS.S	PPC440GP style cache	(no cache control registers; must use TLB)
CH602CAS.S	MPC602 style cache	(restricted *HID0* control; I-cache enabled)
CH603CAS.S	MPC603 style cache	(full *HID0* control)
CH860CAS.S	MPC860 style cache	(special control registers)
CH8560CA.S	MPC8560 style cache	(special control registers)

The following table summarizes the supported cache types and identifies which type should be used for the various PowerPC implementations.

Processor	Function	*icsize*	*icparam*	*dcsize*	*dcparam*
PPC401, 401GF, 401A1, 401x2, 401B3, PPC403, 403GA, 403GB, 403GC	*ch403cache*	*2048*	*0x20010*	*1024*	*0x20010*
PPC403GCX (see note 2)	*ch403cache*	*16384*	*0x20010*	*8192*	*0x20010*
PPC405, 405GP, 405B3	*ch405cache*	*16384*	*0x20020*	*8192*	*0x20020*
PPC440GP	*ch440cache*	*32768*	*0x400020*	*32768*	*0x400020*
MPC505, 509	*ch860cache*	*4096*	*16*	*0*	*0*
MPC555	none				
MPC5553, MPC5554 (see note 4)	*ch5553cache*	*0*	*0*	*0*	*0*
MPC601 (see note 3)					
MPC602	*ch602cache*	*4096*	*32*	*4096*	*32*
MPC603	*ch603cache*	*8192*	*32*	*8192*	*32*
MPC603e, MPC8240, 8260, 8280	*ch603cache*	*16384*	*32*	*16384*	*32*
MPC604	*ch603cache*	*16384*	*32*	*16384*	*32*
MPC604e (see note 2)	*ch603cache*	*32768*	*32*	*32768*	*32*
MPC5200	*ch603cache*	*16384*	*32*	*16384*	*32*
MPC8349	*ch603cache*	*32768*	*32*	*32768*	*32*
MPC740, 750, MPC7400	*ch603cache*	*32768*	*32*	*32768*	*32*
MPC801, 821, 823, 850, 855, 860	*ch860cache*	*4096*	*16*	*4096*	*16*
MPC8560	*ch8560cache*	*32768*	*32*	*32768*	*32*

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. For example, you must use this technique to accommodate the PPC403GCX or MPC604e.

Note 3: The MPC601 has a 32K unified instruction and data cache which can only be manipulated by adjusting the *IBAT* registers. For this reason, no AMX cache control function is provided for the MPC601.

Note 4: The MPC5553 has an 8K unified instruction and data cache.
The MPC5554 has a 32K unified instruction and data cache.
Cache control function *ch5553cache* automatically determines the available cache size. The cache parameters are unused by this function.

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 403 Cache Services

The PPC401 and PPC403 families have instruction and data caches which are manipulated using control bits in the special purpose control registers provided for that purpose. All processors with caches of this type use the AMX type 403 cache services. The PPC403 family uses its *ICCR* and *DCCR* cacheability registers 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 PPC403 cache support module *CH403CAS.S* includes an initialization function *ch403ccr* which is used to specify the *ICCR* and *DCCR* masks which define your cached memory regions. Function *ch403ccr* MUST be called before any other cache service function is used. Its prototype is as follows:

```
void CJ_CCPP ch403ccr(unsigned long iccr, unsigned long dccr);
```

Parameter *iccr* is a bit mask defining the bits in the *ICCR* register which are to be cleared (set) to disable (enable) the instruction cache.

Parameter *dccr* is a bit mask defining the bits in the *DCCR* register which are to be cleared (set) to disable (enable) the data cache.

The cache control function *ch403cache* is used to 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 invalidated. If the data cache is selected, it is disabled, flushed if previously enabled and then 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.

Parameter *icparam* is used to encode two 16-bit parameters: the number of instruction cache sets (ways) and the number of bytes in each instruction cache line. For the PPC403 family with 2-way instruction cache and 16 bytes per cache line, *icparam* is *(2<<16)+16*.

Parameter *dcparam* is used to encode two 16-bit parameters: the number of data cache sets (ways) and the number of bytes in each data cache line. For the PPC403 family with 2-way data cache and 16 bytes per cache line, *dcparam* is *(2<<16)+16*.

The board initialization function *chbrdinit* in the 403 EVB board support module *CH403EVB.S* illustrates the proper use of the PPC403 cache control functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. Function *chbrdinit* calls *ch403ccr* to define the *ICCR* and *DCCR* masks for the 403 EVB board.

Chbrdinit also includes a call to *ch403cache* to initialize the PPC403 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 *CH403EVB.S* to allow the call to *ch403cache*.

Type 405 Cache Services

The PPC405 family has instruction and data caches which are manipulated using control bits in the special purpose control registers provided for that purpose. All processors with caches of this type use the AMX type 405 cache services. The PPC405 family uses its `ICCR` and `DCCR` cacheability registers 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 PPC403 and PPC405 caches are managed in exactly the same manner. However, the PowerPC special purpose register used for cache control is different for each of these PowerPC families. For this reason, the description of the "Type 403 Cache Services" applies equally to the PPC405 family with the following minor differences. Functions `ch405ccr` and `ch405cache` replace functions `ch403ccr` and `ch403cache` respectively. The board initialization function `chbrdinit` for the PPC405GP Reference Board is provided in board support module `CH405EVB.S`.

Type 602 Cache Services

The MPC602 instruction and data cache are manipulated using control bits in the `HID0` register. Both caches can be invalidated but only the data cache can be disabled.

The cache control function `ch602cache` is used to 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 invalidated without being disabled. If the data cache is selected, it is flushed if previously enabled and then it is disabled and invalidated. The data cache, if selected, is 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` define the number of bytes in each instruction and data cache line respectively.

For an example of the MPC602 memory and cache initialization, refer to the discussion of the Ultra 603 board initialization function `chbrdinit` in the next section. Use it as an example and replace references to `ch603cache` with `ch602cache`.

Type 603 Cache Services

The MPC603, MPC604, MPC7xx, MPC7400, MPC5200, MPC8240, MPC8260, MPC8280 and MPC8349 families have instruction and data caches which are manipulated using control bits in the *HID0* register. All processors with caches of this type use the AMX type 603 cache services. Both caches can be invalidated and disabled or enabled. The data cache can be flushed.

The cache control function *ch603cache* is used to 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 invalidated. If the data cache is selected, it is flushed if previously enabled and then it is disabled and 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* define the number of bytes in each instruction and data cache line respectively.

When using the MPC603, you must, prior to launching AMX, ensure that the MMU is properly initialized to condition the instruction and data block address translation registers (*IBATn* and *DBATn*) to meet your hardware memory addressing specifications and caching requirements. You must also condition the machine state register (*MSR*) to enable/disable block address translation for instructions (*MSR* bit *IR*) and/or data (*MSR* bit *DR*).

You will also have to condition the hardware implementation register (*HID0*) to globally enable/disable the instruction cache (*HID0* bit *ICE*) and/or data cache (*HID0* bit *DCE*).

The AMX PPC32 Ultra 603 board support module *ULT603.S* contains a board initialization procedure *chbrdinit* which sets the *IBATn* and *DBATn* registers to allow access to the devices and memory connected to the PCI or ISA buses and to permit the software *IACK* command to be used with the Intel 8259 Interrupt Controller. Procedure *chbrdinit* also sets the *IBATn* and *DBATn* registers to support instruction and data caching. However, the procedure does not modify register *HID0* which, by default on the Ultra 603, is set to enable instruction caching and disable data caching for all such cached memory regions.

The board initialization function *chbrdinit* in the Ultra 603 board support module *ULT603.S* illustrates the proper use of the MPC603 cache control function. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* also includes a call to *ch603cache* to initialize the MPC603 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 *ULT603.S* to allow the call to *ch603cache*.

If you are not using the MPC603, be sure to use the board support module for the board which you are using. For example, if you are using the MPC8260 on the EST SBC8260 board, use board support module *SBC8260.S* in place of *ULT603.S*.

Type 860 Cache Services

The MPC8xx and MPC50x families have instruction and data caches which are manipulated using control bits in the special purpose control registers provided for that purpose. All processors with caches of this type use the AMX type 860 cache services. Both caches can be invalidated and disabled or enabled. The data cache can be flushed. Special control bits are available to force the data cache to operate in write-through mode and to lock the instruction and/or data cache content.

The cache control function *ch860cache* is used to 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 invalidated. If the data cache is selected, it is disabled, flushed if previously enabled and then 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* define the number of bytes in each instruction and data cache line respectively.

Most AMX applications will execute with the instruction and data caches fully operational. The data cache may have to operate in write-through mode and both caches will usually need to be unlocked. The MPC860ADS board used by KADAK was tested in this fashion. The cache control function *ch860cache* includes the support necessary to adjust these cache mode parameters. The allowable bit masks for the *command* parameter are extended as follows:

0x00000008L	Force data cache to operate in write-through mode. (Only if the data cache is selected in *command*)
0x00000004L	Unlock the selected caches.

The board initialization function *chbrdinit* in the MPC860ADS board support module *CH860ADS.S* illustrates the proper use of the MPC860 cache initialization functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* also includes a call to *ch860cache* to initialize the MPC860 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 unlocked, initialized and left disabled, edit source file *CH860ADS.S* to allow the call to *ch860cache*.

If you are not using the MPC860, be sure to use the board support module for the board which you are using. For example, if you are using the MPC823 on the RPX Lite MPC823 board, use board support module *CH823RPX.S* in place of *CH860ADS.S*.

Type 8560 Cache Services

The MPC85xx family has separate Level 1 (L1) instruction and data caches and a Level 2 (L2) unified cache.

Type 8560 Level 1 Caches

The L1 instruction and data caches are manipulated using control bits in the special purpose control registers provided for that purpose. All processors with caches of this type use the AMX type 8560 cache services. Both caches can be invalidated and disabled or enabled. The data cache can be flushed. Special instructions are available to lock the instruction and/or data cache content.

The cache control function *ch8560cache* is used to 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 invalidated. If the data cache is selected, it is flushed if previously enabled and then it is disabled and 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* define the number of bytes in each instruction and data cache line respectively.

Most AMX applications will execute with the instruction and data caches fully operational. Both caches will usually need to be unlocked. The MPC8560ADS board used by KADAK was tested in this fashion. The cache control function *ch8560cache* includes the support necessary to adjust this cache mode parameter. The allowable bit masks for the *command* parameter are extended as follows:

0x00000004L	Unlock the selected caches.

Type 8560 Level 2 Cache

The L2 unified cache is manipulated using control bits in a memory mapped register provided for that purpose. The cache can be invalidated and disabled or enabled. Since the cache operates in a write-through fashion, it does not need to be flushed.

The L2 cache may be split for use as a cache with a separate SRAM block or it may be used entirely as an SRAM block. Normally, the L2 cache will cache all memory accesses. However, it may be configured to cache only instruction or only data accesses.

To maintain coherency between the processor core, the L1 caches, the L2 cache and external memory, the *ABE* bit of the *HID1* special purpose register must be set whenever the L2 cache is enabled. Special instructions are available to lock the L2 instruction and/or data cache content.

The L2 cache control function *ch8560cache2* is used to configure, invalidate, unlock and enable/disable the L2 cache as indicated by its *command* parameter.

```
void CJ_CCPP ch8560cache2(unsigned int command,
                          unsigned long srambase);
```

Command parameters (merge one or more of these bit masks):

0x80000000L	Configure the L2 cache/SRAM
0x00000004L	Unlock the cache
0x00000001L	*0/1* = disable/enable the cache

Command configuration parameters:

(merge one of the following four values)

0x00000000L	Disable entire L2 cache/SRAM block
0x00010000L	Configure block as all L2 cache
0x00020000L	Configure block as all SRAM *srambase* specifies the address of the SRAM block
0x00030000L	Configure block as split L2 cache/SRAM *srambase* specifies the address of the SRAM block

(merge one of the following three values if L2 cache is used)

0x00000000L	L2 cache is instruction and data cache
0x00200000L	L2 cache is data cache only
0x00400000L	L2 cache is instruction cache only

If the configure command is not selected by parameter *command*, the L2 cache is invalidated and then unlocked and enabled/disabled as directed by parameter *command*.

If the configure command is selected by parameter *command*, the L2 cache/SRAM is unconditionally disabled, invalidated and unlocked and then reconfigured as specified. The cache is then enabled if so directed by parameter *command*.

If the specified configuration enables access to an SRAM block, parameter *srambase* specifies the base address of the SRAM memory. When used, *srambase* must be aligned to a 256K boundary. If *srambase* is *0xFFFFFFFF*, then the SRAM block base address is not changed from its previously configured value.

MPC8560 Cache Initialization

When using the MPC8560, you must, prior to launching AMX, ensure that the MMU is properly initialized to condition the TLB table entries to meet your hardware memory addressing specifications and caching requirements.

The board initialization function *chbrdinit* in the MPC8560ADS board support module *ADS8560.S* illustrates the proper use of the MPC8560 cache initialization functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* also includes calls to *ch8560cache* and *ch8560cache2* to initialize the MPC8560 L1 and L2 caches. Note that the calls, although present in *chbrdinit*, are actually skipped so that caches are not altered unless you so desire. If you want the caches to be unlocked, initialized and left disabled, edit source file *ADS8560.S* to allow the calls to *ch8560cache* and *ch8560cache2*.

Type 440 Cache Services

The PPC440GP family has instruction and data caches which are manipulated using cache control instructions. The caches of type 440 targets are always enabled. The only way to control the cacheability of a region of memory is with the "caching inhibited" bit of the TLB entry which controls access to the memory region. The TLB entries must be initialized by you to match your memory system and to meet the needs of your application.

The cache control function *ch440cache* is used to flush and invalidate the instruction and/or data caches as indicated by its *command* parameter. If the instruction cache is selected by parameter *command*, it is invalidated. If the data cache is selected, it is flushed and then invalidated. The selected caches remain enabled regardless of parameter *command*.

Parameters *icsize* and *dcsize* define the total instruction and data cache sizes respectively.

Parameter *icparam* is used to encode two 16-bit parameters: the number of instruction cache sets (ways) and the number of bytes in each instruction cache line. For the PPC440GP family with 64-way instruction cache and 32 bytes per cache line, *icparam* is *(64<<16)+32*.

Parameter *dcparam* is used to encode two 16-bit parameters: the number of data cache sets (ways) and the number of bytes in each data cache line. For the PPC440GP family with 64-way data cache and 32 bytes per cache line, *dcparam* is *(64<<16)+32*.

The board initialization function *chbrdinit* in the PPC440GP board support module *CH440EVB.S* illustrates the proper use of the PPC440GP cache control function. Function *chbrdinit* is called from the AMX Sample Program *main* function.

Chbrdinit also includes a call to *ch440cache* to initialize the PPC440GP 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 invalidated, edit source file *CH440EVB.S* to allow the call to *ch440cache*.

Type 5553 Cache Services

The MPC555x family has a unified instruction and data cache which is manipulated using control bits in the special purpose control registers provided for that purpose. All processors with caches of this type use the AMX type 5553 cache services. The cache can be flushed and invalidated and enabled or disabled. Special control bits are available to control cache locking and select write-through or copy-back write mode. Cache locking and write mode must be configured by you in your board initialization code prior to starting your AMX application. The AMX type 5553 cache services do not alter the cache locking and write mode control bits.

The cache control function `ch5553cache` is used to flush, invalidate and enable/disable the unified cache as indicated by its command parameter. The read-only cache configuration register provides the necessary cache size information. The cache is unconditionally flushed and invalidated. The cache is then enabled or disabled as directed by parameter `command`. The branch target prediction buffer is invalidated and enabled.

Parameters `icsize`, `dcsize`, `icparam` and `dcparam` are not used by the AMX type 5553 cache services and should be set to zero.

When using any of the MPC555x processors, you must, prior to starting your AMX application, ensure that the MMU is properly initialized to condition the TLB table entries to meet your hardware memory addressing specifications and caching requirements.

The board initialization function `chbrdinit` in the MPC5553 board support module `MPC5553.S` illustrates the proper use of the MPC5553 cache initialization functions. Function `chbrdinit` is called from the AMX Sample Program `main` function. `Chbrdinit` also includes a call to `ch5553cache` to initialize the MPC5553 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 flushed and invalidated, edit source file `MPC5553.S` to allow the call to `ch5553cache`.

D.3 Customizing AMX Cache Services

Unfortunately, the PowerPC Architecture does not include a definition of how caches should be implemented and manipulated. Consequently, an increasing number of PowerPC processors are emerging, each with different approaches to cache management.

The cache services provided by AMX accommodate the different cache control mechanisms employed by IBM and Motorola in their PowerPC products. However, new variants continue to emerge at regular intervals.

At the time AMX PPC32 was first released, there were four cache management schemes in use. Most new PowerPC processors will use one of these existing methods 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 the PPC403GCX processor, adjust the custom cache parameters as follows.

ch403cache	Cache function name	(in AMX Library)
16384	Instruction cache size	(16K bytes)
0x20010	Instruction cache parameter	(2-way; 16 bytes/line)
8192	Data cache size	(8K bytes)
0x20010	Data cache parameter	(2-way; 16 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 ch403cache,16384,0x20010,8192,0x20010
```

> Note
>
> The cache parameters in the *...CACHE* directive must be provided in a form acceptable to the PowerPC 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	(16 bytes/line)
8192	Data cache size	(8K bytes)
16	Data cache parameter	(16 bytes/line)

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 *command* parameter will adhere to the standard bit mask values supported by all *chXXXcache* functions (see Appendix D.2).

The interpretation of the remaining four cache control parameters by your procedure *YOURcache* must be as follows.

The parameter *icsize* must define the total size, in bytes, of the instruction cache. The least significant 16 bits of parameter *icparam* must define the number of bytes in each instruction cache block (cache line). The upper 16 bits are free for interpretation by your cache control function.

The parameter *dcsize* must define the total size, in bytes, of the data cache. The least significant 16 bits of parameter *dcparam* must define the number of bytes in each data cache block (cache line). The upper 16 bits are free for interpretation by your cache control function.

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
```

D.4 Cache Problems with MMU

The Problem

If the PowerPC Memory Management Unit (MMU) is used, care must be taken to avoid a very subtle exception handling problem which can occur if the memory system is not properly managed. Since device interrupts generate exceptions, the problem cannot be ignored.

Stated simply, AMX PPC32 requires that the virtual memory system established by the MMU **must match** the default real memory system configuration which exists when the MMU is not active. Access to all regions of memory (code, data and I/O) on which AMX and your application depend must appear not to change while AMX is executing.

Unfortunately, the PowerPC architecture definition makes meeting this requirement difficult. When a processor exception occurs, most PowerPC implementations unconditionally clear the *IR* and *DR* control bits in the Machine Status Register (*MSR*) to disable instruction and data address translation by the MMU. Consequently, the exception begins service with the default, processor dependent real memory access attributes in effect. If these attributes do not match the virtual memory attributes that were in effect prior to the exception, catastrophic failures can be expected.

Obviously, your virtual memory space must map the processor Exception Vector Table to the correct real memory region or else instruction fetches will fail the instant the exception occurs. That fault is so obvious that it will have to be repaired before your application can even run.

Since the processor automatically disables address translation when an exception occurs, device I/O operations may not operate correctly. Without address translation, device reads or writes may be cached or even issued out of sequence.

For most PowerPC implementations, when an exception occurs, the instruction and data caches are enabled by default unless otherwise specified by cache control flags in some processor dependent registers. This enabling of the data cache can also lead to the erroneous operation of an AMX exception handler. The following example, only one of many possible scenarios, will show how such a fault can occur.

Suppose that you are operating with the MMU providing address translation such that the data region in which all AMX and task stacks reside is cache inhibited. When an exception occurs, AMX saves a few registers on the stack. The values end up cached because address translation has been disabled and the data cache is considered enabled. AMX may then adjust the *MSR* to once again enable address translation so that the data region is again cache inhibited. Eventually, AMX tries to restore the saved registers and the restoral may fail. Why? Since the saved values were cached, their restoral will cause a cache hit. However, if a memory reference generates a cache hit when the cache is disabled, the result is not predicable. In fact, the PowerPC specification refers to such an access as a programming error.

The improper operation of AMX exception handlers is rarely encountered. The reason is that, for most processors, the MMU and cache control registers can be properly managed to prevent the error conditions just described. For example, when an MPC860 exception occurs, the instruction and data caches are not arbitrarily enabled. Their state is dictated by cache control settings in the *MI_CTR* and *MD_CTR* registers. Hence, by proper programming of the MMU and cache control registers (as will be described), cache problems in exception handlers can be avoided.

The Solution

AMX provides two features which can be used to avoid problems in exception handlers. In most cases, a simple adjustment of the *MSR* content after the exception occurs is all that is needed to restore proper MMU and cache operation. In other cases, the adjustment must be done without allowing caching to occur while registers are saved.

The ***MSR* adjustment** when an exception is serviced restores one or more of the *MSR* control bits to the state which existed prior to the exception. For example, the MSR may have to be adjusted to restore (set) the *IR* and *DR* control bits to allow address translation so that device I/O operations will operate correctly without caching side effects.

The *MSR* adjustment is an AMX target configuration option. Use the AMX Configuration Manager to edit your AMX Target Parameter File. The *MSR* adjustment mask is entered on the EVT Service property page. Refer to the description of the MSR adjustment parameter in Chapter 4.2.

AMX also offers an **alternate register saving strategy** which avoids memory caching when register values are saved on entry to an exception handler. When the alternate strategy is invoked, AMX saves three general purpose registers in three of the special purpose registers (SPRs) reserved by the PowerPC architecture for operating system use. By default, AMX saves its registers in SPRs designated as *SPRG0*, *SPRG1* and *SPRG2*.

The alternate register strategy is only of use when an *MSR* adjustment is required and the MMU and cache cannot be initialized to avoid the caching side effects previously described. To incorporate this strategy into your AMX application, you must use a text editor and edit your AMX Target Parameter File to include the following directive.

```
...MSRX
```

It is recommended that the directive be inserted immediately following the *...MSR* directive which defines the *MSR* adjustment. When this form of the *...MSRX* directive is used, AMX will use SPR 272 (*SPRG0*), 273 (*SPRG1*) and 274 (*SPRG2*) for temporary storage of the registers while it adjusts the *MSR*.

KADAK has observed that some debug monitors (such as the monitor on the Motorola Ultra 603 Motherboard Platform) use one or more reserved SPRs for their operation. To avoid conflict with such software, you may be able to adapt AMX to use a different set of special purpose registers by using the following form of the *...MSRX* directive.

```
...MSRX  273,274,275
```

In this example, AMX will use SPR 273 (*SPRG1*), 274 (*SPRG2*) and 275 (*SPRG3*) for temporary storage. Any three SPRs which are free for dedicated use by the operating system or your application can be used. The order of the SPR register numbers in the list does not matter. However, when this form of the directive is used, all three parameter values must be provided.

MMU and Type 403 and 405 Caches

When a PowerPC like the PPC403 or PPC405 generates an exception, the *IR* and *DR* control bits in the *MSR* are cleared, disabling the MMU. Cache operation is then governed by register pairs such as the *ICCR* and *DCCR* registers which determine if instruction and data caching is to be enabled or disabled for each of 32 specific regions of memory. Other register pairs define the default memory access attributes of those 32 memory regions.

If the settings in the default data cache control registers which govern the RAM used for AMX and task stacks match the TLB descriptions which apply to that RAM when the MMU is enabled, AMX exception handlers will operate without error. The AMX *MSR* adjustment can be used to alter the *MSR* upon entry to the exception handler if required.

If the default cache control register settings cannot be made to match the TLB page table descriptions, you must restore the *IR* and *DR* control bits in the *MSR* to enable the MMU for proper memory caching. To do so, you must use the alternate register saving strategy to prevent caching errors when registers are saved for use by AMX.

You must maintain **coherency** between your MMU and cache control settings. Hence you must always ensure that the settings of the default cache control registers always match the TLB descriptions whenever the MMU is enabled. The procedure is simplified if the regions of memory controlled by the MMU correspond exactly to the 32 regions specified by the default cache control registers.

For example, if the data cache for a region is normally enabled and you plan to disable the data cache for that region, you must:

1) disable interrupts to prevent any interrupt exception,
2) disable the MMU,
3) revise the TLB description to disable the data cache for the region,
4) invalidate (flush) the data cache for the region,
5) clear the control bit for the memory region in the *DCCR* register,
6) enable the MMU and
7) enable interrupts.

MMU and Type 602/603 Caches

When a PowerPC like the MPC602 or MPC603 generates an exception, the *IR* and *DR* control bits in the *MSR* are cleared, disabling the MMU. Cache operation for ALL of memory is then assumed to be governed by a *WIMG* value of *0011* which specifies write-back, cache enabled, multi-processor, guarded memory operations. The level 1 cache will only be enabled if permitted by the settings of the *ICE* and *DCE* control bits in the *HID0* register. Level 2 cache operations may be further qualified by settings in other cache control registers such as the *L2CR* register in the MPC7400.

If this mode of operation upon entry to the exception handler matches the memory access rules for the RAM used for AMX and task stacks prior to the exception (as established by the *DBATn* or page table register settings), then AMX exception handlers will operate without error. The AMX *MSR* adjustment can be used to alter the *MSR* upon entry to the exception handler if required.

If there is a change in the rules for accessing the RAM used for AMX and task stacks, you must restore the *IR* and *DR* control bits in the *MSR* to enable the MMU for proper memory caching. To do so, you must use the alternate register saving strategy to prevent caching errors when registers are saved for use by AMX.

You must maintain **coherency** between your MMU and cache control settings. Hence you must always ensure that the settings of the *ICE* and *DCE* control bits in the *HID0* register always match the *IBATn* and *DBATn* or page table register settings whenever the MMU is enabled.

For example, if the data cache is normally enabled and you plan to disable the data cache, you must:

1) disable interrupts to prevent any interrupt exception,
2) disable the MMU,
3) revise the *DBATn* or TLB descriptions to disable the data cache,
4) invalidate (flush) the data cache,
5) clear the *DCE* control bit in the *HID0* register,
6) enable the MMU and
7) enable interrupts.

MMU and Type 860 Caches

When a PowerPC like the MPC860 generates an exception, the *IR* and *DR* control bits in the *MSR* are cleared, disabling the MMU. Cache operation for ALL of memory is then governed by the cache inhibit default (*CIDEF*) bit setting in the *MI_CTR* and *MD_CTR* cache control registers.

If the default data cache operation upon entry to the exception handler matches the memory access rules for the RAM used for AMX and task stacks prior to the exception (as established by the MMU page table register settings), then AMX exception handlers will operate without error. The AMX *MSR* adjustment can be used to alter the *MSR* upon entry to the exception handler if required.

If there is a change in the rules for accessing the RAM used for AMX and task stacks, you must restore the *IR* and *DR* control bits in the *MSR* to enable the MMU for proper memory caching. To do so, you must use the alternate register saving strategy to prevent caching errors when registers are saved for use by AMX.

You must maintain **coherency** between your MMU and cache control settings. Hence you must always ensure that the cache inhibit default (*CIDEF*) bit setting in the *MI_CTR* and *MD_CTR* cache control registers always match the MMU page table register settings whenever the MMU is enabled.

For example, if the data cache is normally enabled and you plan to disable the data cache, you must:

1) disable interrupts to prevent any interrupt exception,
2) disable the MMU,
3) revise the MMU page table descriptions to disable the data cache,
4) invalidate (flush) the data cache,
5) set the *CIDEF* control bit in the *MD_CTR* register,
6) enable the MMU and
7) enable interrupts.

MMU and Type 5553, 8560 and 440 Caches

When a PowerPC like the MPC5553, MPC8560 or PPC440GP generates an exception, the *IS* and *DS* control bits in the *MSR* are cleared and the state of the instruction and data caches does not change. Cache operation is then governed by the MMU TLB table entries that match Instruction Space (*IS*) *0* and Data Space (*DS*) *0*.

If the data cache access configuration for Data Space *0* upon entry to the exception handler matches the memory access rules for the RAM used for AMX and task stacks prior to the exception (as established by the MMU TLB table entries and the *MSR IS* and *DS* bits), then AMX exception handlers will operate without error. The AMX *MSR* adjustment can be used to alter the *MSR* upon entry to the exception handler if required.

If there is a change in the rules for accessing the RAM used for AMX and task stacks, you must restore the *IS* and *DS* control bits in the *MSR* to use the correct address spaces for proper memory caching. To do so, you must use the alternate register saving strategy to prevent caching errors when registers are saved for use by AMX.

For the MPC8560 and PPC440GP you must maintain **coherency** between your MMU and cache control settings. Hence you must always ensure that the settings of the cache control registers match the TLB descriptions at all times.

For example, if the data cache for a region is normally enabled and you plan to disable the data cache for that region, you must:

1) disable interrupts to prevent any interrupt exception,
2) revise the TLB description to disable the data cache for the region,
3) invalidate (flush) the data cache for the region and
4) enable interrupts.

This page left blank intentionally.