



## **AMX™ MA32 Target Guide**

**First Printing: June 1, 2001**  
**Last Printing: March 1, 2005**

**Copyright © 2001 - 2005**

**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](mailto:amxtech@kadak.com)

**Copyright © 2001-2005 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. MIPS32 is a trademark of MIPS Technologies, Inc. All other trademarked names are the property of their respective owners.

**AMX MA32 TARGET GUIDE**  
**Table of Contents**

	<b>Page</b>
<b>1. Getting Started with AMX MA32</b>	<b>1</b>
1.1 Introduction .....	1
1.2 AMX Files .....	2
1.3 AMX Nomenclature .....	4
1.4 AMX MA32 Target Specifications .....	5
1.5 Launch Requirements .....	6
<b>2. Program Coding Specifications</b>	<b>11</b>
2.1 Task Trap Handler .....	11
2.2 Task Scheduling Hooks .....	12
<b>3. The Processor Interrupt System</b>	<b>13</b>
3.1 Operation .....	13
3.2 AMX Vector Table .....	17
3.3 AMX Interrupt Priority .....	20
3.4 Conforming ISPs .....	22
3.5 Nonconforming ISPs .....	25
3.6 Processor Vector Initialization .....	27
<b>4. Target Configuration Module</b>	<b>29</b>
4.1 The Target Configuration Process .....	29
4.2 Target Configuration Parameters .....	33
4.3 Interrupt Service Procedure (ISP) Definitions .....	49
4.4 Defining a Fast Clock ISP .....	56
4.5 Null Functions .....	58
4.6 ROM Option Parameters .....	59
<b>5. Clock Drivers</b>	<b>61</b>
5.1 Clock Driver Operation .....	61
5.2 Custom Clock Driver .....	63
5.3 AMX Clock Drivers .....	65
5.3.1 Time Base Clock Driver .....	65
5.3.2 8254 Clock Driver .....	67

## AMX MA32 TARGET GUIDE

### Table of Contents (Cont'd)

Appendices	Page
<b>Appendix A. Target Parameter File Specification</b>	<b>A-1</b>
A.1 Target Parameter File Structure .....	A-1
A.2 Target Parameter File Directives .....	A-3
A.3 Porting the Target Parameter File .....	A-16
<b>Appendix B. AMX MA32 Service Procedures</b>	<b>B-1</b>
<b>Appendix C. AMX MA32 ROM Option</b>	<b>C-1</b>
<b>Appendix D. Cache Management</b>	<b>D-1</b>
D.1 AMX Cache Services .....	D-1
D.2 Low Level Cache Control Services .....	D-4
D.3 Customizing AMX Cache Services .....	D-9
<b>Appendix E. Interrupt Management</b>	<b>E-1</b>
E.1 Interrupt Prioritization and Nesting .....	E-1
E.2 Nested Interrupts .....	E-3
E.3 Interrupt Identification Procedure for Nested Interrupts .....	E-4

## AMX MA32 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 AMX Exception Handlers .....	13
Figure 3.2-1 AMX Vector Table .....	17
Figure 3.3-1 AMX Interrupt Priorities .....	21
Figure 3.6-1 Exception Vector Code Fragments .....	28
Figure 4.1-1 Configuration Manager Screen Layout .....	30
Figure A.1-1 AMX Target Parameter File .....	A-1

# 1. Getting Started with AMX MA32

## 1.1 Introduction

The AMX™ Multitasking Executive is described in the AMX User's Guide. This target guide describes AMX MA32 which operates in 32-bit mode on all MIPS processors which comply with the MIPS32™ Architecture specification.

Throughout this manual, the term MIPS32 refers specifically to the MIPS Technologies MIPS32 family of processors and all processors which are MIPS32 architecture compliant. When distinctions are not important, the term MIPS32 is used to reference any processor which has the general MIPS32 characteristics. 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 MA32 real-time system. It is assumed that you have read the AMX User's Guide and are familiar with the architecture of the MIPS32 processor.

### Note

AMX MA32 only operates in 32-bit mode. AMX MA32 does NOT support the use of 64-bit memory systems, registers or instructions.

## Installation

AMX MA32 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 MA32 has been tested by KADAK, a separate chapter in the **AMX MA32 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 MIPS32 assembly language.

Figures 1.2-1, 2 and 3 summarize the AMX modules provided with AMX MA32. 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.

<b>File Name</b>	<b>Module</b>
<i>CJ442 .H</i>	Generic include file
<i>CJ442APP.H</i>	Custom application definitions
<i>CJ442CC .H</i>	C dependent definitions
<i>CJ442EC .H</i>	AMX error code definitions
<i>CJ442IF .H</i>	C and target interface prototypes
<i>CJ442KC .H</i>	Private AMX constants
<i>CJ442KF .H</i>	AMX service procedure prototypes
<i>CJ442KP .H</i>	Private AMX prototypes
<i>CJ442KS .H</i>	Private AMX structure definitions
<i>CJ442KT .H</i>	Target processor definitions
<i>CJ442KV .H</i>	AMX version specification
<i>CJ442SD .H</i>	AMX application structure definitions
<i>CJ442TF .H</i>	Target dependent prototypes
<i>CJZZZ .H</i>	Copy of generic include file <i>CJ442.H</i> used for portability
<i>CHxxxxx .H</i>	Definitions for common timer (PIT) and serial I/O (UART) chips

Figure 1.2-1 AMX Include Files

<b>File Name</b>	<b>Module</b>
<i>CJ442K .DEF</i>	Private AMX assembly language definitions
<i>CJ442ESV.S</i>	AMX Exception Supervisor
<i>CJ442KQ .S</i>	Private AMX math procedures
<i>CJ442KR .S</i>	AMX Interrupt Supervisor
<i>CJ442KS .S</i>	AMX Task Scheduler
<i>CJ442MXA.S</i>	Message Exchange Manager constants
<i>CJ442TDC.S</i>	Time/Date Manager constants
<i>CJ442UA .S</i>	Target processor and C support
<i>CJ442UB .S</i>	Target processor and C support

Figure 1.2-2 AMX Assembler Source Files

<b>File Name</b>	<b>Module</b>
<i>CJ442KA .C</i>	Kernel task services
<i>CJ442KB .C</i>	General task services
<i>CJ442KBR.C</i>	
<i>CJ442KC .C</i>	Timer Manager
<i>CJ442KCR.C</i>	
<i>CJ442KD .C</i>	Task management services
<i>CJ442KDR.C</i>	
<i>CJ442KE .C</i>	Task termination services
<i>CJ442KF .C</i>	Suspend/resume task
<i>CJ442KG .C</i>	Time slice services
<i>CJ442KH .C</i>	Task status
<i>CJ442KI .C</i>	Enter and Exit AMX
<i>CJ442KJ .C</i>	General object access
<i>CJ442KK .C</i>	AMX Vector Table access
<i>CJ442KL .C</i>	Private AMX list manipulation
<i>CJ442KM .C</i>	AMX task scheduler hook services
<i>CJ442KX .C</i>	AMX Kernel Task
<i>CJ442CL .C</i>	Circular List Manager
<i>CJ442LM .C</i>	Linked List Manager
<i>CJ442BM .C</i>	Buffer Manager
<i>CJ442BMR.C</i>	
<i>CJ442EM .C</i>	Event Manager
<i>CJ442EMR.C</i>	
<i>CJ442RM .C</i>	Semaphore Manager (resources)
<i>CJ442SM .C</i>	Semaphore Manager
<i>CJ442SMR.C</i>	
<i>CJ442MB .C</i>	Mailbox Manager
<i>CJ442MBR.C</i>	
<i>CJ442MF .C</i>	Flush mailbox and message exchange
<i>CJ442MM .C</i>	Memory Manager
<i>CJ442MMR.C</i>	
<i>CJ442MX .C</i>	Message Exchange Manager
<i>CJ442MXR.C</i>	
<i>CJ442TDA.C</i>	Time/Date Manager
<i>CJ442TDB.C</i>	Time/Date formatter
<i>CJ442UF .C</i>	Launch and leave AMX
<i>CJ442XTA.C</i>	Message exchange task services
<i>CJ442XTB.C</i>	Message exchange task termination
<i>CHxxxxxxT.C</i>	Clock drivers for common timer (PIT) chips
<i>CHxxxxxxS.C</i>	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:

<i>cjkkpppp</i>	AMX C procedure name <i>pppp</i> for service of class <i>kk</i>
<i>cjxtttt</i>	AMX structure name of type <i>tttt</i>
<i>xttttyyy</i>	Member <i>yyy</i> of an AMX structure of type <i>tttt</i>
<i>CJ_ID</i>	AMX object identifier (handle)
<i>CJ_ERRST</i>	Completion status returned by AMX service procedures
<i>CJ_CCPP</i>	Procedures use C parameter passing conventions
<i>CJ_ssssss</i>	Reserved symbols defined in AMX header files
<i>CJ_ERxxxx</i>	AMX Error Code <i>xxxx</i>
<i>CJ_WRxxxx</i>	AMX Warning Code <i>xxxx</i>
<i>CJ_FExxxx</i>	AMX Fatal Exit Code <i>xxxx</i>
<i>CJ442xxx.xxx</i>	AMX MA32 filenames
<i>CJZZZ.H</i>	Generic AMX include file

The generic include file *CJZZZ.H* is a copy of file *CJ442.H* which includes the subset of the AMX MA32 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 MIPS32 processor.

Processor registers are referenced using the software names specified by the MIPS32 architecture rather than the register numbers.

*zero, at, v0, v1, a0-a3*  
*t0-t7, s0-s7, t8, t9*  
*k0, k1, gp, sp, s8, ra*

## 1.4 AMX MA32 Target Specifications

AMX MA32 was initially developed and tested using the MIPS 4Kc processor on the MIPS Malta Development Board. However, the AMX MA32 design criteria fully encompass the MIPS32 architecture specifications.

AMX uses a set of design constants which vary according to the constraints imposed by each target processor. When operating on the MIPS32 processor, these design constants assume the values listed in Figure 1.4-1.

<b>Symbol</b>	<b>Purpose</b>
<i>CJ_CCISIZE</i>	Size of integer is 4 bytes (32 bits)
<i>CJ_ID</i>	Event group supports 32 event flags per group
<i>CJ_ERRST</i>	AMX id (handle) is a 32 bit unsigned integer AMX error codes are 32 bit signed integers
<i>CJ_MINMSZ</i>	Minimum AMX message size is 12 bytes
<i>CJ_MAXMSZ</i>	Default AMX message size is 12 bytes
<i>CJ_MINKG</i>	Minimum number of AMX message envelopes is 10
<i>CJ_MINKS</i>	Minimum Kernel Stack is 512 bytes
<i>CJ_MINIS</i>	Minimum Interrupt Stack is 512 bytes
<i>CJ_MINTKS</i>	Minimum task storage (including TCB) is 1024 bytes
<i>CJ_MINBFS</i>	Minimum AMX buffer size is 8 bytes
<i>CJ_MINUMEM</i>	Minimum AMX memory block size is 16 bytes
<i>CJ_MINSMEM</i>	Minimum AMX memory section size is 128 bytes

Figure 1.4-1 AMX Design Constants

## 1.5 Launch Requirements

The MIPS32 processor must be properly configured for use before AMX is launched. The manner in which this is accomplished will depend on your target hardware implementation and on the startup code provided with your C compiler.

AMX does not include bootstrap code to initialize the MIPS32 processor. It is assumed that you will have a boot ROM present which configures the MIPS32 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 be set to **kernel mode**. Set field *KSU* to 0 in the processor status register.

Since AMX operates in kernel mode, there is no need to enable **coprocessor 0**. However, good programming practice suggests that, for processor portability, you should enable coprocessor 0 by setting *CU0* to 1 in the processor status register. If you use any other coprocessor, you must enable it by setting *CU1*, *CU2* or *CU3* to 1. Set *CU<sub>n</sub>* to 0 for all unused coprocessors.

All other mode setting bits in the status register must be set to match your specific hardware configuration.

#### Warning

AMX MA32 only operates in 32-bit mode. AMX MA32 does NOT support the use of 64-bit memory systems, registers or instructions. Hence, you must not set bits *KX*, *SX* or *UX* in the processor status register to select 64-bit operation. If you manage to perform a 64-bit operation, the upper 32 bits of the registers will be lost if the executing code is preempted for any reason.

## Interrupt State

Interrupts can be enabled or disabled on entry to AMX. Processor status bits *EXL* and *ERL* should be 0; AMX will reset them to 0. Processor status bit *IE* 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 ( $IM[N] = 0; N = 2 \text{ to } 7$ ). 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.

Software interrupts, controlled by bits *IM[0]* and *IM[1]* in the processor status register, can be enabled or disabled as required by your application. You must not generate a software interrupt request until you have installed an AMX Interrupt Service Procedure to properly service the request.

Once AMX has been launched, you must not alter the processor status register directly. This implies that you cannot use any C library processor support procedures to alter the processor status register. Use AMX procedure *cjcfsrget* to read the status register. Use AMX procedure *cjcfsrset* to alter the status register.

Prior to launching AMX or after AMX has shut down, you can use AMX procedures *cjcfflagrd* and *cjcfflagwrx* to manipulate the content of the processor status register. All other AMX procedures which affect the status register can only be used while AMX is active.

## Exception Vector Location

The location of the MIPS32 exception vectors in memory is dictated by the state of the *BEV* bit in the coprocessor 0 status register. In some implementations, the state of the *BEV* bit may be hardwired, either internally or externally, by the processor. In other cases, the *BEV* bit is set to 1 by a processor reset and then altered by the bootstrap code which services the reset exception.

Prior to launching AMX, the *BEV* bit in the status register must be set to match the exception vector specification provided in your AMX Target Parameter File as described in Chapter 4.2. Once AMX has been launched, the state of the *BEV* bit must not be altered.

If your MIPS32 processor implements the Special Interrupt Exception Vector and you have configured AMX to service it, then you must also set the *IV* bit in the coprocessor 0 cause register prior to launching AMX.

## Instruction and Data Caching

The MIPS32 architecture defines a System Control Coprocessor (CP0) which, if present, can provide for instruction and/or data cache implementation. AMX supports caches controlled by CP0 as defined by the MIPS32 architecture. AMX cache management is restricted to the kernel address space (*kseg0*) controlled by field *κ0* in the configuration register (CP0 register 16).

You must be aware that, on processors which utilize the MIPS32 Translation Lookaside Buffer (TLB) for memory management, successful cache operation may depend on proper setup of the TLB. AMX does not manipulate the TLB. For example, if the TLB 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 TLB is properly initialized to condition the instruction and data address translation logic to meet your hardware memory addressing specifications and caching requirements.

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 MIPS32 family of processors are described in Appendix D.

## Memory Management Unit (TLB)

The MIPS32 architecture defines a System Control Coprocessor (CP0) which, if present and fully implemented, includes a memory management unit. The Translation Lookaside Buffer (TLB) in CP0 provides this feature. AMX does not support the MIPS32 TLB.

If you are using AMX on MIPS32 processors which do not implement TLB addressing, this restriction does not apply. These processors do not implement the MIPS32 memory management unit and allow direct access to the full address space.

Your AMX application code and data must reside within the memory address ranges allowed by the particular MIPS32 processor which you are using. The MIPS32 TLB, 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 TLB for your specific hardware configuration prior to entry to your `main()` program.

Your AMX application code and data must reside in the `kseg0` and `kseg1` virtual address spaces. Code and data references must not use `kuseg`, `kseg2` or `kseg3` virtual addresses since such references require TLB support. Your MIPS32 application is therefore restricted to 512 MBytes of cached code and data in `kseg0` and 512 MBytes of uncached code and data in `kseg1`.

### Warning!

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

## Big or Little Endian

AMX MA32 is delivered ready for use with the little endian model in which the least significant byte of a word (long) is stored in the lowest byte address. AMX MA32 will also operate, without modification, on big endian hardware in which the most significant byte of a word (long) is stored in the lowest byte address. However, to use AMX on big endian hardware, you must first rebuild the AMX Library for big endian operation as described in Appendix D of the AMX User's Guide.

To use the little endian model, you must set `BE` to 0 in the configuration register (`CP0` register 16) prior to launching AMX. To use the big endian model, you must set `BE` to 1. The state of `BE` is usually set by hardware when the processor is reset.

This page left blank intentionally.

## 2. Program Coding Specifications

### 2.1 Task Trap Handler

Arithmetic overflow is the only exception generated by the MIPS32 processor for which a task trap is provided. An overflow trap occurs if the processor executes an *ADD*, *ADDI*, *SUB* or *SUBI* instruction that generates a two's complement overflow.

The Task Trap Handler can be written as a C procedure with formal parameters.

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

void CJ_CCPP traphandler(
    struct cjsxregs *regp,          /* A(Register structure) */
    CJ_VOIDPROC *faultpp,         /* A(Fault pointer)      */
    CJ_TYREG causereg)           /* Cause register content */
{
    :
    Process the error
    :
}
```

The arithmetic overflow exception trap is serviced by AMX. The state of each register at the time of the fault is stored on the stack in an AMX register structure *cjsxregs*. Parameter *regp* is a pointer to that structure. Structure *cjsxregs* is defined in AMX header file *CJ442KT.H*.

Interrupts are enabled upon entry to the task trap handler. Note that the status register copy in the register array reflects the state of the status register after the exception occurred.

A copy of the cause register at the time of the exception is provided as parameter *causereg*. Cause register bit *BD* (bit 31) indicates whether or not the instruction which caused the fault occurred in a branch delay slot.

A pointer to the address of the fault is provided as parameter *faultpp*. If bit *BD* in *causereg* is 0, the void procedure pointer at *\*faultpp* points to the instruction which caused the fault. If bit *BD* in *causereg* is 1, the fault occurred in a delay slot, and *\*faultpp* is the address of the branch instruction immediately preceding the instruction that caused the fault.

The register values in structure *regs* can be examined and, in rare circumstances, modified. If necessary, the procedure pointer at *\*faultpp* can be modified, with extreme care, to force resumption at some other location in the task code. If the task trap handler returns to AMX, execution will resume at the location specified by *\*faultpp* with registers set according to the values in the structure referenced by *regp*.

Since the task trap handler executes in the context of the task in which the exception occurred, it is free to use all AMX services normally available to tasks. In particular, the handler can call *cjtkend* to end task execution if so desired.

## 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 disabled and must remain so.
- The stack pointer in register *sp* references the task's stack.
- The Task Control Block address is in register *a1*.
- The return address is in register *ra*.
- Registers *at*, *v0*, *v1*, *a0*, *a1*, *a2* and *a3* 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 *CJ442K.DEF* in your assembly language module, you can reference the private region within the TCB reserved for your use as *XTCBUSER(a1)*.

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

### 3. The Processor Interrupt System

#### 3.1 Operation

The MIPS32 architecture classifies all internal and external sources of interruption as exceptions. The processor automatically vectors control to one of several unique exception addresses within the processor address space and begins execution at that address. These exception vector addresses are dictated by the MIPS32 architecture and vary according to the settings of *BEV* in the processor status register and *IV* in the processor cause register.

The fragment of code which resides in memory at a particular exception vector address is called an **exception handler**. Upon entry to the handler, all internal, external and software interrupts are inhibited and the processor is operating in kernel mode (*EXL=1* or *ERL=1* in the processor status register). The reason for the exception is recorded in the processor cause register. The address of the fault or the location at which processing should resume is contained in one of two exception program counter registers in CP0 (*EPC* if *EXL=1* or *ErrorEPC* if *ERL=1*). By convention, registers *k0* and *k1* are free for use by the exception handler. Other CP0 registers may also contain exception specific information.

AMX provides exception handlers for the exception vectors indicated in Figure 3.1-1. For the supported exceptions, those which will be serviced by AMX will be determined by the selections in your AMX Target Parameter File (see Chapter 4.2).

Exception	AMX handler	BEV=0	BEV=1
Reset, soft reset, NMI	none	0xBFC0.0000	0xBFC0.0000
TLB refill ( <i>EXL=0</i> )	<i>cj_kpextlb</i>	0x8000.0000	0xBFC0.0200
XTLB refill (64-bit addresses)	none	0x8000.0080	0xBFC0.0280
Cache error	<i>cj_kpexcer</i>	0xA000.0100	0xBFC0.0300
General	<i>cj_kpexgen</i>	0x8000.0180	0xBFC0.0380
Special interrupt ( <i>IV=1</i> )	<i>cj_kpexsiv</i>	0x8000.0200	0xBFC0.0400
Debug	<i>cj_kpexdbg</i>	0xBFC0.0480 (not dependent on <i>BEV</i> )	

Figure 3.1-1 AMX Exception Handlers

## AMX Exception Handler Operation

Each AMX exception handler services all of the exceptions which are trapped through a particular processor exception vector. The AMX cache error exception handler (*cj\_kpexcer*) and debug exception handler (*cj\_kpexdbg*) simply loop endlessly, preventing further execution. All other AMX exception handlers determine the cause of the exception and then branch via the address specified by an entry in the AMX Vector Table to an appropriate exception specific procedure. Collectively, the AMX exception handlers are referred to as the AMX Exception Supervisor.

Upon entry to any exception specific procedure, the processor state will match its state at the time of the exception. Registers *k0* and *k1* are free for use. All other registers must be preserved. Register *k1* contains the AMX vector number for the particular exception as defined in Figure 3.2-1. The procedure executes on the stack that was in effect at the time of the exception.

The AMX Configuration Builder can build an exception service root, a small shell of code written in assembly language which calls an application function (usually coded in C) to service the event. The root code saves a subset of processor registers, calls your function, restores the registers and, if applicable, resumes execution from the point at which the exception occurred. The service root makes writing the procedure to handle an exception a very simple process.

### Note

The AMX cache error and debug exception handlers do not permit further execution. The handlers simply plug these complex exception vectors for systems not equipped to handle cache errors or hardware debug traps.

## Interrupt Service Procedures

There are eight exceptions dedicated to internal or external hardware interrupts 0 to 5 and software interrupts 0 and 1. The particular procedure which services one of these interrupt exceptions is called an Interrupt Service Procedure (ISP). The ISP must service the interrupt source and dismiss the interrupt request. The ISP root of a conforming AMX Interrupt Service Procedure, to be described in Chapter 3.4, is a special prebuilt ISP provided by AMX to simplify interrupt service.

In the case of software interrupts 0 or 1, the software interrupt request will have already been cleared (*IP[0]* or *IP[1]* set to 0 in the cause register) by the AMX exception handler.

If the AMX exception handler is unable to determine the source of an interrupt request (*IP[0..7]=0* in the cause register), it treats the exception as an undefined interrupt and services it through an entry in the AMX Vector Table reserved for that purpose.

## Exception Service Procedures

The particular procedures which service non-interrupt exceptions are referred to as exception service procedures. Each such procedure must deal with the specific exception as required by your application. In general, there is little that can be done to service an exception since most are just signals of abnormal processor behavior.

AMX lets you ignore an exception (rarely advisable), treat the event as a fatal condition or pass the exception through to the exception handler which was in place prior to launching AMX. If an exception is passed through to the prior handler, the machine state will be as defined by the exception being serviced. If an exception is ignored or treated as fatal, the condition which led to the exception is not handled (cleared or otherwise serviced) by AMX. However, unless passed through to the prior handler, software interrupts 0 or 1 are always cleared by AMX.

Note that the AMX exception handler for the general exception vector treats the arithmetic overflow exception as a special case. Unless otherwise specified by your AMX configuration, AMX uses its task trap mechanism to service this exception. If the exception occurs within an executing task which has provided a task trap handler, then that task's trap handler is called as described in Chapter 2.1. Otherwise, the exception is treated as a fatal exception generated by a task trap.

At any time, your application can install a pointer to a custom exception service procedure into any entry in the AMX Vector Table, including entries normally used for interrupt service. Your procedure will then be executed whenever the AMX exception handler identifies that exception as the one to be serviced.

The ISP root of a nonconforming AMX Interrupt Service Procedure, to be described in Chapter 3.5, is a special prebuilt exception service root provided by AMX to simplify exception and nonconforming interrupt service.

## Fatal Exception Procedure

Unless otherwise specified by your AMX Target Parameter File (see Chapter 4.2), AMX will treat an exception as fatal if you have not provided an Interrupt Service Procedure or exception service procedure for that exception. In such a case, the AMX exception handler calls a Fatal Exception Procedure *cjksfatalexh* in module *CJ442UF.C* identifying the exception and the machine state at the time of the exception.

If the Fatal Exception Procedure returns, AMX calls its standard Fatal Exit Procedure *cjksfatal* (also in module *CJ442UF.C*) with one of the following AMX fatal exit codes:

<i>CJ_FETRAP</i>	Fatal exception
<i>CJ_FEISPTRAP</i>	Task trap in ISP
<i>CJ_FETKTRAP</i>	Task trap occurred: in a Restart Procedure or in a Timer Procedure or in a task with no task trap handler

The Fatal Exception Procedure *cjksfatalexh* is written in C as follows. Upon entry, interrupts are in the state determined by the particular exception.

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

void CJ_CCPP cjksfatalexh(
struct cjxregs *regp, /* A(Register structure) */
int faultid, /* Fault identifier */
CJ_TYREG causereg) /* Cause register content */
{
:
Process the error
:
}
```

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 *CJ442KT.H*.

The address of the fault, as determined by the content of the *EPC* or *ErrorEPC* register at the time of the exception, is provided in the register array at *regp->xreg\_sp*.

A copy of the cause register at the time of the exception is provided as parameter *causereg*. Cause register bit *BD* (bit 31) indicates whether or not the address at which the fault occurred is in a branch delay slot.

The *faultid* is the AMX vector number *CJ\_PRVNxxxx* (see Figure 3.2-1) identifying the exception.

## 3.2 AMX Vector Table

AMX maintains a Vector Table containing pointers to service procedures for all of the interrupts and exceptions which the processor and AMX can generate. The order of entries in the Vector Table is illustrated in Figure 3.2-1. The vector numbers are defined in AMX header file *CJ442KT.H*. The vector number mnemonic *CJ\_PRVNxxxx* must be used in all calls to AMX *cjksixxxx* procedures to identify entries in the AMX Vector Table.

Vector Name	Vector Number	Exception
<i>CJ_PRVNUND</i>	0	Undefined external interrupt (generated by AMX)
<i>CJ_PRVNMOD</i>	1	TLB modification
<i>CJ_PRVNTLBL</i>	2	TLB miss - load ( <i>EXL=1</i> )
<i>CJ_PRVNTLBS</i>	3	TLB miss - store ( <i>EXL=1</i> )
<i>CJ_PRVNADEL</i>	4	Address error - load
<i>CJ_PRVNADES</i>	5	Address error - store
<i>CJ_PRVNIIBE</i>	6	Bus error - instruction
<i>CJ_PRVNDIBE</i>	7	Bus error - data
<i>CJ_PRVNSYS</i>	8	<i>SYSCALL</i> instruction trap
<i>CJ_PRVNBPF</i>	9	Breakpoint instruction trap
<i>CJ_PRVNRI</i>	10	Reserved instruction
<i>CJ_PRVNCPU</i>	11	Coprocessor Unusable
<i>CJ_PRVNOVF</i>	12	Arithmetic Overflow
<i>CJ_PRVNTR</i>	13	Trap
<i>CJ_PRVNVCEI</i>	14	Virtual Coherency Exception (Instruction)
<i>CJ_PRVNFPP</i>	15	Floating Point Trap
	16,17	Reserved
<i>CJ_PRVNC2E</i>	18	Coprocessor 2 Exception
	19,20,21	Reserved
<i>CJ_PRVNMDMX</i>	22	MDMX Coprocessor Unusable
<i>CJ_PRVNWPP</i>	23	Watchpoint
<i>CJ_PRVNMC</i>	24	Machine Check
	25..30	Reserved
<i>CJ_PRVNVCEI</i>	31	Virtual Coherency Exception (Data)
<i>CJ_PRVNSW0</i>	32	Software interrupt 0
<i>CJ_PRVNSW1</i>	33	Software interrupt 1
<i>CJ_PRVNIN0</i>	34	Hardware interrupt 0
<i>CJ_PRVNIN1</i>	35	Hardware interrupt 1
<i>CJ_PRVNIN2</i>	36	Hardware interrupt 2
<i>CJ_PRVNIN3</i>	37	Hardware interrupt 3
<i>CJ_PRVNIN4</i>	38	Hardware interrupt 4
<i>CJ_PRVNIN5</i>	39	Hardware interrupt 5
<i>CJ_PRVNTLBR</i>	40	TLB refill ( <i>EXL=0</i> )
<i>CJ_PRVNMUX</i>		Base of vectors for multiplexed interrupts (if any)

Figure 3.2-1 AMX Vector Table

## Multiplexed Interrupts

The MIPS32 architecture provides support for only six hardware interrupt sources. Many evaluation boards and custom applications require support for a much larger number of interrupting devices. In such cases, multiple interrupts are funneled through a single MIPS32 hardware interrupt exception and software must be provided to determine the interrupt source and service the device accordingly.

AMX provides support for this type of multiplexed interrupt handling. For each multiplexed hardware interrupt exception, AMX reserves a block of  $n$  vectors in the AMX Vector Table. These blocks of vectors are located starting at vector number *CJ\_PRVNMUX* in the table. The order of the vector blocks and the number of vectors in each is specified in your AMX Target Parameter File as described in Chapter 4.2.

When a multiplexed hardware interrupt occurs, the AMX exception handler calls an application Interrupt Identification Procedure (IIP) to identify the specific interrupting device. AMX then dispatches through one of the vectors in the vector block in the AMX Vector Table to a device specific Interrupt Handler.

## Interrupt Identification Procedure

When multiple devices generate interrupts through a single hardware interrupt exception, you must provide an Interrupt Identification Procedure (IIP) which the AMX exception handler can call to determine the device which actually generated the interrupt. The IIP for each multiplexed interrupt exception is defined by you in your AMX Target Parameter File (see Chapter 4.2). The IIP must be coded in MIPS32 assembly language.

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

Interrupts are enabled ( $IE=1$  and  $ERL=EXL=0$ ) but interrupts are masked off according to the interrupt priority established by your definition of the multiplexed hardware interrupt exception.

The processor is executing in kernel mode.

The return address is in register *ra*.

The stack pointer in register *sp* references the AMX Interrupt Stack.

Registers *at*, *v0* and *v1* are free for use.

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 particular hardware interrupt exception.

The interrupt number is returned in register *v0* as follows:

$v0 = i$	Interrupt number ( $0$ to $n-1$ )
$v0 = -1$	Ignore the interrupt
$v0 = -2$	Generate a fatal exception (unidentified interrupt request)

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

If your IIP returns to AMX with  $v0 = -2$ , AMX generates a fatal exception using the AMX vector number  $CJ\_PRVNIi$  ( $i = 0$  to  $5$ ) to identify the multiplexed hardware interrupt exception through which the spurious interrupt occurred.

Note

If an interrupt controller is used, the IIP can provide interrupt prioritization and nesting. Guidelines for doing so are provided in Appendix E.

### 3.3 AMX Interrupt Priority

The MIPS32 family of processors does not offer inherent interrupt priority ordering. However, the AMX Interrupt Supervisor implements this feature and allows the nesting of interrupts for fast response to high priority events.

AMX establishes an interrupt ordering in which  $IP[7]$  is the highest priority and  $IP[0]$  is the lowest priority. Software interrupts 1 and 0 ( $IP[1]$  and  $IP[0]$ ) are therefore the very lowest in priority, even if used to simulate device interrupts. The AMX priority ordering is not adjustable.

To achieve this interrupt priority ordering, AMX maintains a private interrupt disable mask which it uses to inhibit all interrupts below a particular priority level. The disable mask is a 32-bit mask which mirrors the interrupt mask, field  $IM$  in the processor status register. If bit  $IM[n]$  is set in the AMX disable mask, AMX will disable the corresponding interrupt source by resetting the corresponding  $IM[n]$  bit in the processor status register to 0.

AMX also maintains a software copy of the considered state of the underlying  $IM$  bits in the processor status register. It must do so in order to be able to restore the correct state of each interrupt enable bit in the processor status register whenever changes in the AMX disable mask occur.

Once AMX has been launched, you must not alter the processor status register directly. This implies that you cannot use any C library processor support procedures to alter the processor status register.

Use AMX procedure `cjcfdmget` to read the AMX disable mask.

Use AMX procedure `cjcfsrget` to read the considered state of the status register.

Use AMX procedure `cjcfsrset` to alter the AMX disable mask and/or the considered state of the status register.

Figure 3.3-1 shows the disable masks for each AMX priority level. Note that tasks execute at the lowest interrupt priority since all hardware and software interrupt requests are enabled (unmasked).

Software interrupts are rarely masked off because they are usually used for software dispatching. If you use a software interrupt to simulate a hardware interrupt, you must adjust all disable masks for higher priority interrupts to mask off that particular software interrupt as well.

<b>Priority Level</b>	<b>Disable Mask</b>	<b>IM Bit</b>	<b>Exception</b>
(highest)	<i>0xFC00</i>	<i>IM[7]</i>	Hardware interrupt 5 (or CP0 count/compare timer)
	<i>0x7C00</i>	<i>IM[6]</i>	Hardware interrupt 4
	<i>0x3C00</i>	<i>IM[5]</i>	Hardware interrupt 3
	<i>0x1C00</i>	<i>IM[4]</i>	Hardware interrupt 2
	<i>0x0C00</i>	<i>IM[3]</i>	Hardware interrupt 1
	<i>0x0400</i>	<i>IM[2]</i>	Hardware interrupt 0
	<i>0x0200</i>	<i>IM[1]</i>	Software interrupt 1
(lowest)	<i>0x0100</i>	<i>IM[0]</i>	Software interrupt 0
	<i>0</i>		AMX tasks

Figure 3.3-1 AMX Interrupt Priorities

### 3.4 Conforming ISPs

A conforming ISP consists of an ISP root and a device Interrupt Handler. The ISP root is created in your Target Configuration Module by the AMX Configuration Generator using the information provided in your Target Parameter File (see Chapter 4.3).

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

The ISP root is the actual Interrupt Service Procedure which is executed by the AMX exception handler when the interrupt occurs.

The ISP root calls the AMX Interrupt Supervisor to switch to the Interrupt Stack, if necessary. The Interrupt Supervisor also adjusts the AMX interrupt disable mask to reflect the priority of the interrupt being serviced.

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

Interrupt Handlers can be written as assembly language or C procedures with or without a single 32-bit formal parameter. The parameter, if needed, is identified in your ISP definition in your Target Parameter File. Use assembly language if speed of execution is critical.

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

- Interrupts are enabled ( $IE=1$  and  $ERL=EXL=0$ ) but interrupts are masked off according to the interrupt priority established by your ISP definition.
- The processor is executing in kernel mode.
- Your Interrupt Handler parameter is in register *v0*.
- The return address is in register *ra*.
- The stack pointer in register *sp* references the AMX Interrupt Stack.
- Registers *at*, *v0* and *v1* are free for use.
- All other registers must be preserved.

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

- Interrupts are enabled ( $IE=1$  and  $ERL=EXL=0$ ) but interrupts are masked off according to the interrupt priority established by your ISP definition.
- The processor is executing in kernel mode.
- Your Interrupt Handler parameter is in register *a0*.
- The return address is in register *ra*.
- The stack pointer in register *sp* references the AMX Interrupt Stack.
- All registers which C treats as alterable are free for use.
- All other registers must be preserved.

The following examples illustrate how simple an Interrupt 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 Interrupt Handler is named *deviceh*.  
The device generates hardware interrupt 2.  
The Interrupt Handler does not require a parameter.

The Interrupt Handler is coded in C as follows:

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

The following C code fragment in a Restart Procedure or task could be used to install the ISP root pointer into the AMX Vector Table.

```
extern void CJ_CCPP deviceisp(void);
:
:
cjsivtwr(CJ_PrvNIN2, (CJ_ISPPROC)deviceisp);
```

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 Interrupt Handler is named *dcb\_ih*.

The device generates hardware interrupt 4.

*deviceXdcb* is the name of the public structure variable which contains information about the specific device.

The Interrupt Handler is coded in C as follows:

```
void CJ_CCPP dcb_ih(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.
    :
}
```

The following C code fragment in a Restart Procedure or task could be used to install the ISP root pointer into the AMX Vector Table, saving the previous ISP root pointer.

```
extern void CJ_CCPP dcb_isp(void);
CJ_ISPPROC oldisproot;
:
:
cjksivtx(CJ_PRVNIN4, (CJ_ISPPROC)dcb_isp, &oldisproot);
```

### 3.5 Nonconforming ISPs

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

Since nonconforming ISPs bypass the AMX Interrupt Supervisor, they cannot make use of the interrupt priority ordering provided by AMX. Consequently, nonconforming ISPs must run with interrupts disabled because they cannot adjust the AMX disable mask or manipulate the processor status register without interfering with AMX. Hence, nonconforming ISPs for interrupt exceptions cannot be nested.

Upon entry to a nonconforming ISP, the processor state matches its state at the time of the interrupt. The processor is in kernel mode with interrupts inhibited ( $EXL=1$  and/or  $ERL=1$  in the processor status register). Registers  $k0$  and  $k1$  are free for use unless one exception has preempted service of another. All other registers must be preserved. The ISP executes on the stack that was in effect at the time of the interrupt.

The nonconforming ISP **must not** permit interrupts to occur.

The nonconforming ISP must dismiss the interrupt request and return to the point of interruption with the *eret* instruction. The *eret* instruction sets  $EXL=0$  (or  $ERL=0$ ) in the processor status register and returns to the point of interruption as specified by CP0 register *EPC* (or *ErrorEPC*).

The most compelling reason to create a nonconforming ISP is to quickly service a device interrupt request which requires little processing and no interaction with AMX. Usually such an ISP is coded in MIPS32 assembly language as illustrated below. An application Restart Procedure or task must install a pointer to the handler (*NONCISP*) into the AMX Vector Table using AMX procedure *cjksivtwr* or *cjksivtx*.

```
.globl          NONCISP
NONCISP:
    : Save only the registers which will be used.
    : Dismiss the interrupt request and service the device.
    : Restore the saved registers.
    eret                               /* Return from exception    */
```

A nonconforming ISP can also be coded in C. Your AMX Target Parameter File (see Chapter 4.3) must include an ISP definition for a nonconforming ISP. The definition provides the name of the ISP root and identifies your C function. An application Restart Procedure or task must install a pointer to the ISP root into the appropriate entry in the AMX Vector Table using AMX procedure *cjksivtwr* or *cjksivtx*.

The C function must be programmed to match the specification of the AMX Fatal Exception Procedure presented in Chapter 3.1. All parameters and execution conditions are identical. If your C function returns to the ISP root, all registers will be restored from the register array and the exception will be dismissed with an *eret* instruction.

## **Exception Service Procedure Implementation**

An exception service procedure is considered to be a special type of nonconforming ISP. It must execute with interrupts inhibited. It cannot make use of AMX services to interact with your application tasks.

Furthermore, since a processor fault can always generate an exception, nesting of exceptions can occur. Hence, a nonconforming ISP can appear nested when preempted by an exception other than one generated by an interrupt.

An exception service procedure can be coded in assembly language or C. Follow the guidelines previously presented for implementing a nonconforming ISP to service an interrupt generated exception.

## 3.6 Processor Vector Initialization

The MIPS32 processor unconditionally jumps to a predetermined memory address whenever an exception occurs. The code located at that address is called an exception handler.

The memory at the exception address can be either ROM or RAM. Your Target Parameter File (see Chapter 4.2) defines whether or not AMX is allowed to modify the code fragment at that address.

If the exception vectors supported by AMX are located in RAM, AMX will, if allowed, install code fragments at those locations to branch to the AMX exception handlers identified in Figure 3.1-1. AMX will alter the first 16 bytes at these locations. The original contents are saved and restored upon exit from AMX.

If you intend to commit a particular exception vector to ROM or if you choose to inhibit AMX from installing its code fragment to support that vector, then you must embed a code fragment at the vector location to branch to the appropriate AMX exception handler identified in Figure 3.1-1. Typical code fragments are illustrated on Figure 3.6-1.

```

.extern  cj_kpexgen      /* AMX General Exception Handler */
.extern  cj_kpexsiv     /* AMX Special Interrupt Handler */
.extern  cj_kpextlb     /* AMX TLB Refill Handler */
.extern  cj_kpexcer     /* AMX Cache Error Handler */
.extern  cj_kpexdbg     /* AMX Debug Exception Handler */

/* General Exception Handler */
/* Locate at address 0x80000180 if BEV = 0 */
/* Locate at address 0xBFC00380 if BEV = 1 */
    la      k0,cj_kpexgen /* k0 = A(handler) */
    jr      k0           /* Go to AMX */
    nop                    /* Delay slot unused */

/* Special Interrupt Exception Handler (if IV=1 in Cause register) */
/* Locate at address 0x80000200 if BEV = 0 */
/* Locate at address 0xBFC00400 if BEV = 1 */
    la      k0,cj_kpexsiv /* k0 = A(handler) */
    jr      k0           /* Go to AMX */
    nop                    /* Delay slot unused */

/* TLB Refill Exception Handler */
/* Locate at address 0x80000000 if BEV = 0 */
/* Locate at address 0xBFC00200 if BEV = 1 */
    la      k0,cj_kpextlb /* k0 = A(handler) */
    jr      k0           /* Go to AMX */
    nop                    /* Delay slot unused */

/* Cache Error Exception Handler */
/* Locate at address 0xA0000100 if BEV = 0 */
/* Locate at address 0xBFC00300 if BEV = 1 */
    la      k0,cj_kpexcer /* k0 = A(handler) */
    jr      k0           /* Go to AMX */
    nop                    /* Delay slot unused */

/* Debug Exception Handler */
/* Locate at address 0xBFC00480 */
    la      k0,cj_kpexdbg /* k0 = A(handler) */
    jr      k0           /* Go to AMX */
    nop                    /* Delay slot unused */

```

Figure 3.6-1 Exception Vector Code Fragments

## 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, *CJ442CM.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.

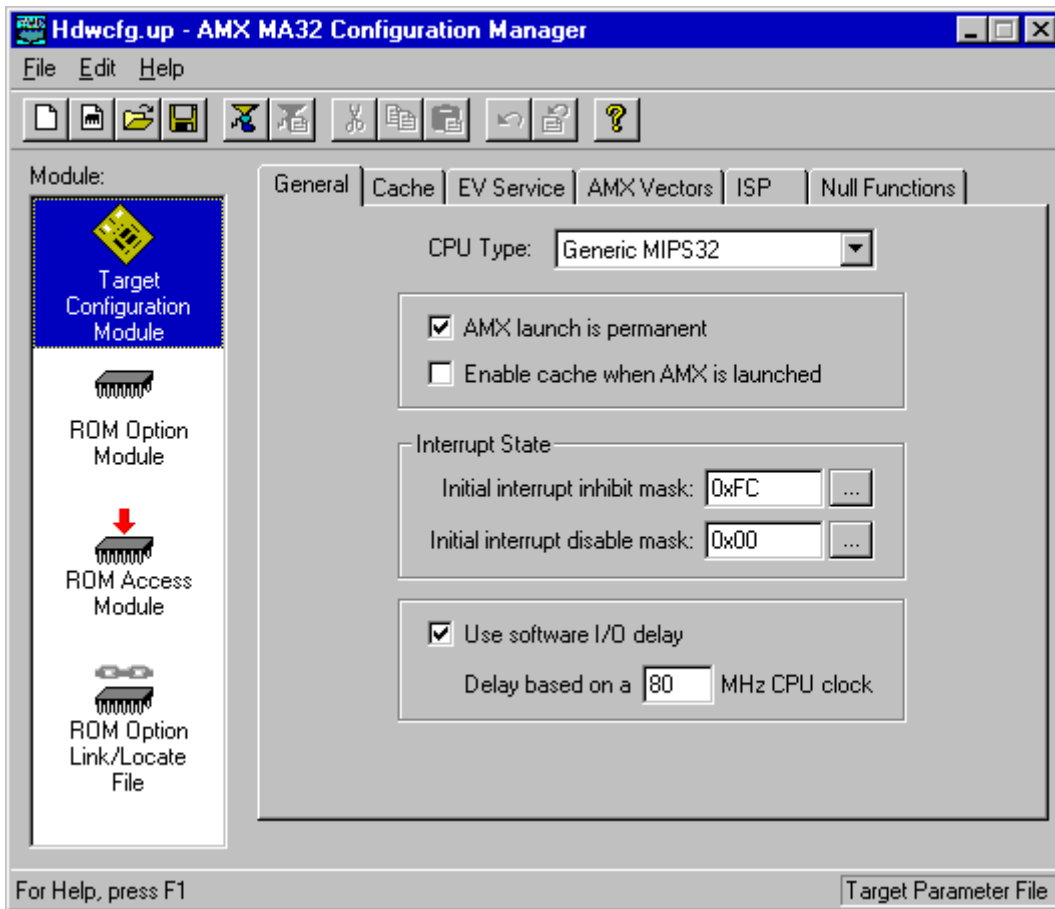


Figure 4.1-1 Configuration Manager Screen Layout

## 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:

<i>Generic MIPS32,</i>	{MIPS32 architecture compliant}
<i>4Kc, 4Km, 4Kp,</i>	{MIPS 4K cores or equivalent}
<i>5K</i>	{MIPS 5K core 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.

## Interrupt State

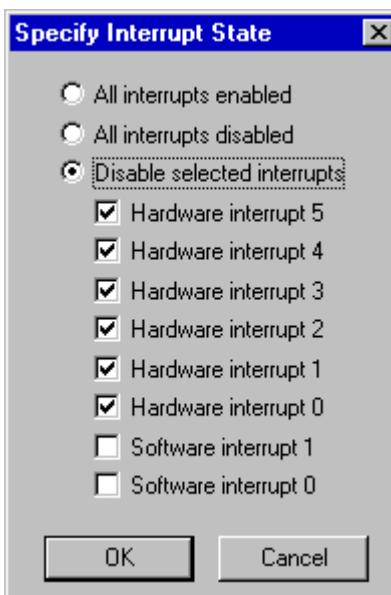
When AMX is launched, it will establish the considered state of the 8-bit interrupt mask (field *IM*) in the processor status register. Thereafter, AMX adjusts the interrupt mask (*IM*) to preclude interrupts identified by the AMX interrupt disable mask. By doing so, AMX ensures that the AMX interrupt priority level is properly maintained.

Set the **Initial interrupt inhibit mask** to define the interrupts which are always to be inhibited when AMX is running at task level. You can use this parameter to permanently inhibit an unused interrupt source until your application decides to enable the source, if ever, with a call to AMX procedure *cjcfsrset()*. Edit the mask value directly or use the [...] button to access the interrupt state dialog window shown below.

Set bit *i* (*i=0* to *7*) to 0/1 to allow/inhibit interrupt request *i*. Mask bits 0 and 1 correspond to software interrupts 0 and 1. Mask bits 2 to 7 correspond to hardware interrupts 0 to 5. The default value of *0xFC* inhibits all hardware interrupts and allows both software interrupts.

Set the **Initial interrupt disable mask** to establish the default value of the AMX interrupt disable mask when AMX is running at task level. AMX will disable the interrupts specified by this mask even if the initial interrupt inhibit mask indicates that the interrupt is normally to be allowed. Since tasks typically run with all hardware and software interrupts enabled, this mask value is usually *0x00*. You can use this parameter to disable an interrupt source until your application alters the mask value with a call to AMX procedure *cjcfsrset()*. Edit the mask value directly or use the [...] button to access the interrupt state dialog window shown below.

When AMX is launched, AMX disables the interrupt system by resetting bit *IE* in the status register to 0. AMX then adjusts the status register interrupt mask (*IM*) to enable or disable each interrupt source according to the specification of your initial interrupt inhibit mask. AMX then selectively disables those interrupts specified by your initial interrupt disable mask. Prior to calling your Restart Procedures, AMX enables the interrupt system by setting bit *IE* in the status register to 1.



## 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.

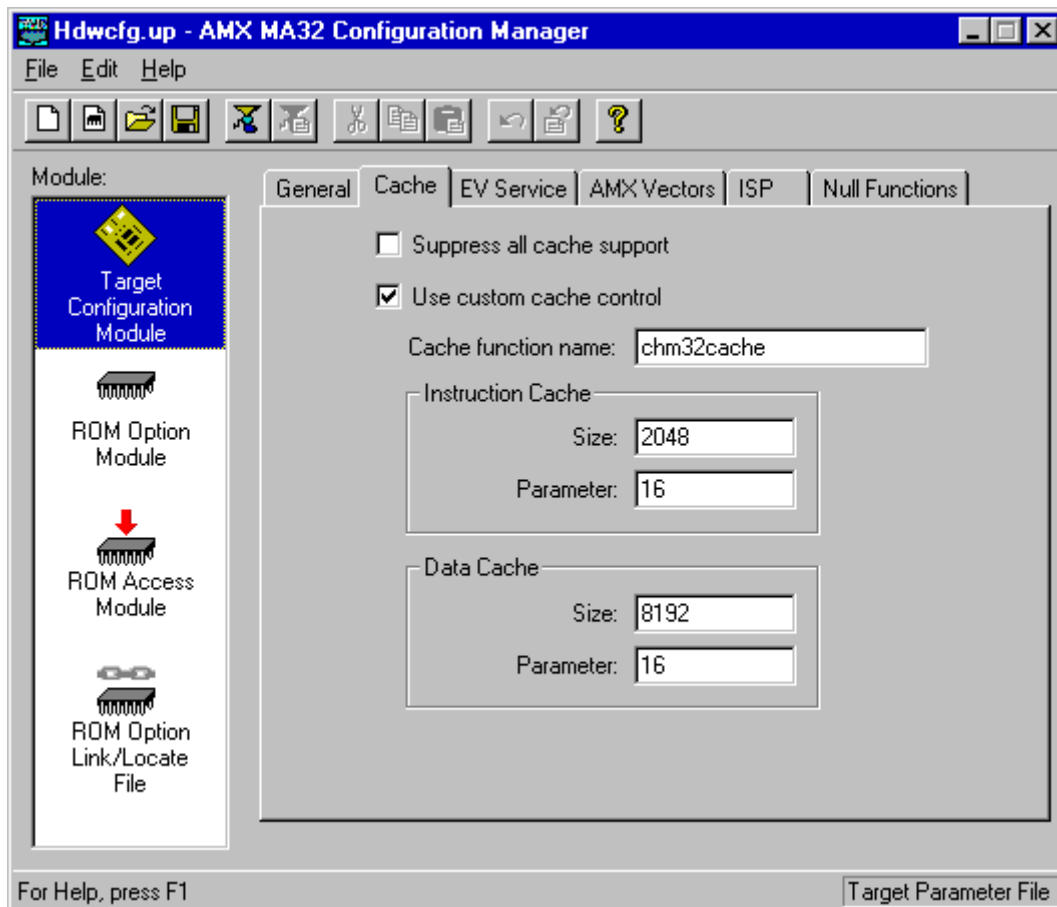
Leave this box unchecked if you want the I/O delay procedure `cjcfhwdelay` to produce no delay beyond that inherent in the procedure call and return.

## 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:

<i>0x80000000L</i>	Select the instruction cache
<i>0x40000000L</i>	Select the data cache
<i>0x0000001L</i>	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:

<i>icsize</i>	Instruction cache size
<i>icparam</i>	Instruction cache parameter
<i>dcsize</i>	Data cache size
<i>dcparam</i>	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 *dcsiz*e to define the total size, in bytes, of the data cache. Parameter *dcp*aram 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

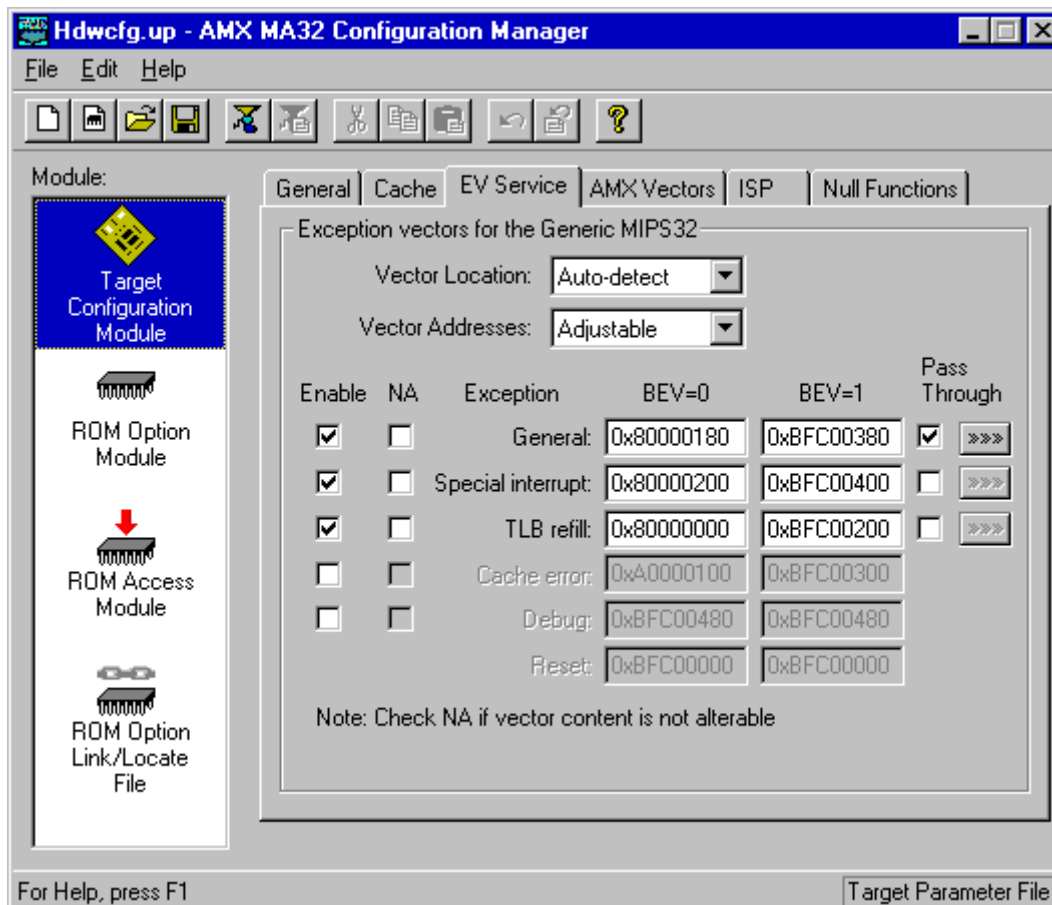
If you provide your own custom cache control function, the interpretation of the four cache control parameters must be as follows.

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 *dcsiz*e must define the total size, in bytes, of the data cache. The least significant 16 bits of parameter *dcp*aram 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.



## Vector Location

You must indicate the default memory location for all processor exception vectors. If you select **Normal**, AMX will assume that the vectors are located at their normal location as would be the case if  $BEV=0$  in the cause register. If you select **Bootstrap**, AMX will assume that the vectors are located at their bootstrap location as would be the case if  $BEV=1$  in the cause register. In either case, AMX will ignore the actual  $BEV$  setting.

If you select **Auto-detect**, AMX will read the  $BEV$  state in the status register when AMX is launched. If  $BEV=0$ , AMX will assume that the vectors are at their normal location. If  $BEV=1$ , AMX will assume that the vectors are at their bootstrap location.

## Vector Addresses

If necessary, you can alter the memory address assigned to any processor exception vector. If you select **Default**, AMX will use the memory addresses specified by the MIPS32 architecture specification. If you select **Adjustable**, the Configuration Builder will allow you to edit the displayed addresses to alter the exception vector locations used by AMX. You might need to do so if you are using a ROM monitor that maintains a shadow set of exception vectors.

## Enable

Check the box for each processor exception vector which you want AMX to support.

## Vectors are Not Alterable (NA)

In most cases, the processor exception vectors will be located in alterable RAM. Therefore, leave the box for each exception vector unchecked.

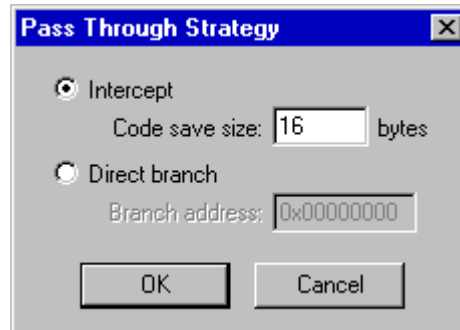
Even if the processor exception vectors will be located in RAM, you can still prevent AMX from altering their content. To do so, check the box for each exception vector which must remain unaltered. In each case, be sure to initialize that exception vector for AMX use as directed in Chapter 3.6.

If your processor exception vectors are in ROM, check the box for each exception vector. In this case, you must initialize the exception vectors in ROM for AMX use as directed in Chapter 3.6.

The NA check box will be ignored if that particular exception vector is not also enabled.

## Pass Through

AMX can pass any of the general exceptions and special interrupts through to the exception handler which was in place prior to launching AMX. To use this feature, check the exception's Pass Through box and then press the button next to it to specify the method of operation. Otherwise, leave the box unchecked.



### Intercept

Check this radio button if AMX must modify the exception vector content to intercept the exceptions and yet remain able to pass some of the exceptions through to the previous handler at that location. Then enter the number of bytes of code that AMX must save from the exception vector location in order to be able to successfully reenter the original handler. Since most handlers begin with a long jump in the first four words, 16 bytes is usually adequate. The save size ( $n$ ) must be 16 or more and a multiple of 4.

Assume that the exception vector is located at address  $EVBASE$ . At launch time, AMX saves  $n$  bytes from address  $EVBASE$  and installs its own 16 byte code fragment to jump to the appropriate AMX exception handler. At the end of its saved copy of the  $n$  bytes, AMX creates a code fragment which uses register  $k0$  to branch to address  $EVBASE+n$ . When any of the exceptions which must be passed through occur, AMX executes its saved copy of the  $n$  bytes. The saved code fragment either jumps directly to the original handler or encounters the code fragment constructed by AMX and branches to address  $EVBASE+n$ .

To determine the save size  $n$ , examine the original code installed at the vector address  $EVBASE$ . Determine how many bytes must be saved so that AMX can successfully reenter the handler. You must save enough instructions so that the code will execute correctly even when relocated and executed elsewhere in memory. Note that register  $k0$  must be free for use if AMX is to be able to branch to address  $EVBASE+n$  to reenter the original handler. If this requirement cannot be met, you must use the Direct Branch option.

The intercept option is not permitted if the exception vector content cannot be modified (its NA box is checked).

### **Direct Branch**

Check this radio button if the exceptions which are to be passed through by AMX must be passed to an exception handler which is located at a particular address in memory. Then enter the memory address at which the handler resides. The address must be a multiple of 4.

You must use the direct branch method if the exception vector content cannot be modified by AMX. You must also use this method if you cannot intercept the original handler and successfully reenter it to pass exceptions through to it.

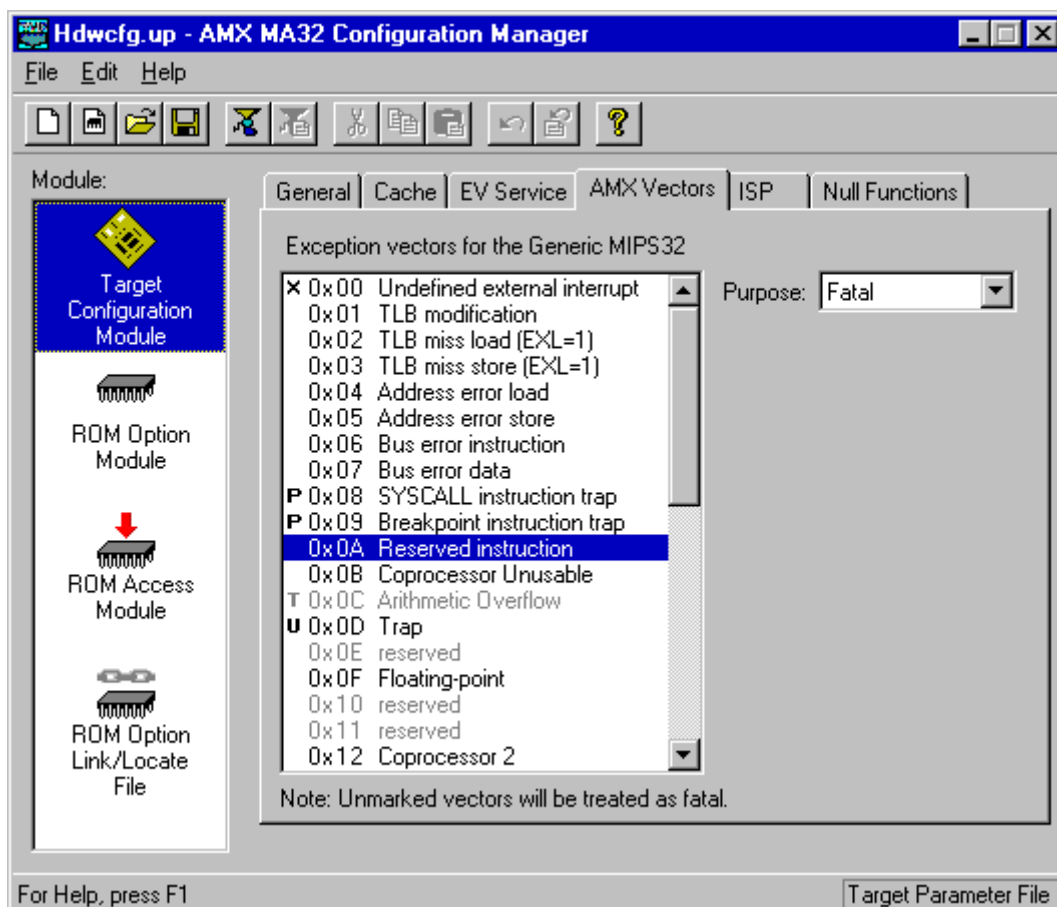
## AMX Vector Definitions

The Target Configuration Module defines the processor exceptions which are to be serviced by AMX. These exceptions are dispatched through entries in the AMX Vector Table. The manner in which each exception is to be serviced is specified by you by selections in the AMX Vector Definition window. The layout of the window is shown below.

The MIPS32 processor selected by the CPU Type on the General Parameters page is identified at the top of the window. The vector list shows all exceptions which AMX supports. The list mirrors the order of entries in the AMX Vector Table. The exceptions which can be generated by the MIPS32 processor 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.

- U** User exception
- T** Task trap
- P** Pass through
- x** Plugged
- M** Multiplexed hardware interrupt
- blank Fatal

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 AMX Vector Table, click on that exception in the vector list. Then, from the pull down list labeled Purpose, choose how you wish to use the exception.

Exceptions that are marked as **User exception** will be passed by AMX to your application Interrupt Service Procedure or exception service procedure. AMX will treat the exception as fatal until you install your service procedure into the AMX Vector Table. If a software interrupt is used, AMX will clear the request prior to dispatching to your handler.

Exceptions that are marked as **Task trap** will be handled using the AMX task trap facility. If the exception occurs while an application task is executing, the exception will be directed to the task's trap handler, if one exists. Otherwise, AMX will treat the exception as fatal.

Exceptions that are marked as **Pass through** will be passed through, unaltered by AMX, to the exception handler which was in place prior to launching AMX.

Exceptions that are marked as **Plugged** will be dismissed by AMX without service. You can use this feature to ignore a spurious, undefined interrupt request which disappears before it can be identified.

Most exceptions should not be plugged. If an internal or external interrupt or software interrupt exception is plugged, your application will appear to hang attempting to service an interrupt request that cannot be removed. Most other exceptions will also recur if the reason for the exception is not first corrected.

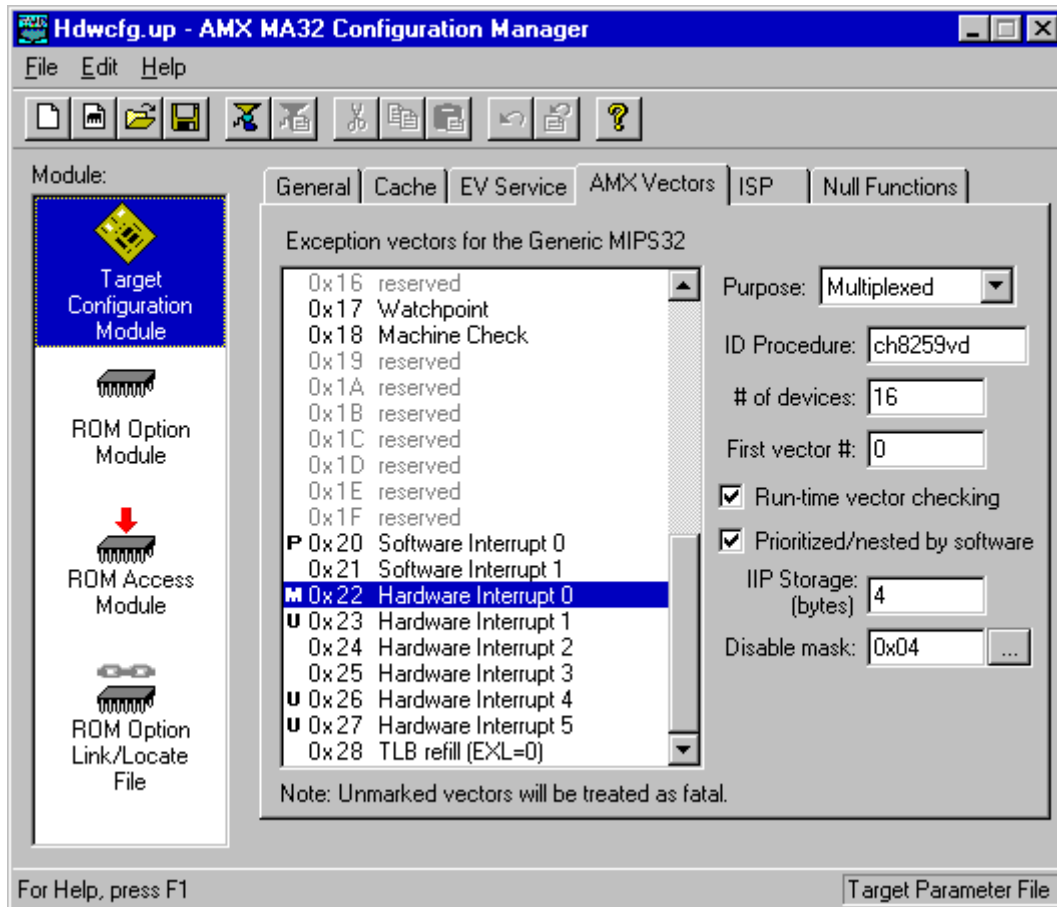
A hardware interrupt exception that is marked as **Multiplexed** will be serviced by an AMX exception handler which will branch to one of  $n$  application Interrupt Service Procedures, where  $n$  is the number of devices which can generate the same hardware interrupt. You must provide an additional set of parameters to specify your multiplexing requirements.

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

## Hardware Interrupt Exceptions

The example illustrated below shows a generic MIPS32 with hardware interrupt 0 being defined for use as a source of multiple AMX device interrupts. Hardware interrupts 1, 4 and 5 are treated as dedicated 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 a hardware exception vector, select User exception as the purpose for the hardware interrupt exception.

## Multiplexed Device Interrupts

If **multiple devices** generate interrupts through the hardware 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 multiplexed 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 occurred via the multiplexed hardware interrupt exception. The AMX vector number  $CJ\_PRVNIN_X$  of the offending hardware 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.

## **Prioritized / Nested by Software**

If you are using an interrupt controller or custom circuitry to manage the interrupt requests from multiple devices, you may be able to enhance your Interrupt Identification Procedure (IIP) to prioritize the interrupts and to allow support for nested interrupts. Instructions for doing so are provided in Appendix E.

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

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

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

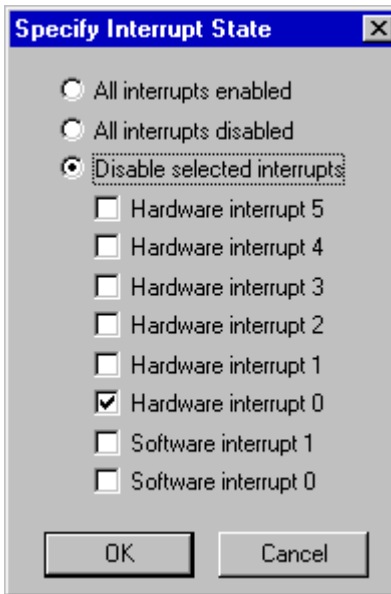
If you leave this box unchecked, nesting of device interrupts on the particular multiplexed hardware interrupt exception will not be supported. AMX will service each interrupt to completion before servicing another interrupt via the same exception.

## Interrupt Disable Mask

AMX services interrupt requests in the order of priority defined in Figure 3.3-1. The ISP root adjusts the AMX interrupt disable mask to ensure that the interrupt is serviced at the correct AMX priority level.

You must identify the hardware and software interrupts which are always to be disabled when servicing the multiplexed hardware interrupt exception. Edit the mask value directly or use the [...] button to access the interrupt state dialog window shown below. Be sure to disable the hardware interrupt being serviced and all other interrupt exceptions deemed to be of lower priority.

Set bit  $i$  ( $i=0$  to  $7$ ) to 0/1 to enable/disable interrupt request  $i$ . Mask bits 0 and 1 correspond to software interrupts 0 and 1. Mask bits 2 to 7 correspond to hardware interrupts 0 to 5.



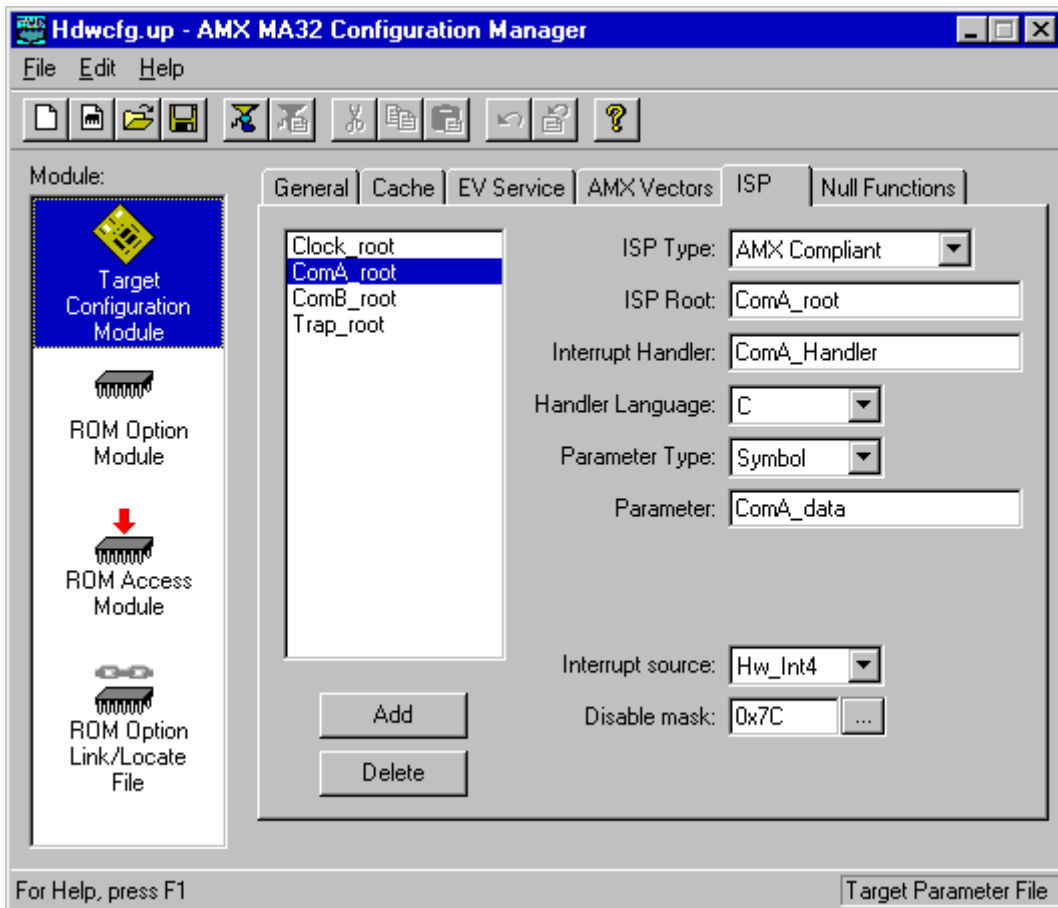
### 4.3 Interrupt Service Procedure (ISP) Definitions

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

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 use the internal MIPS32 timer count/compare registers in CP0 as an AMX clock, choose Time Base Clock from the pull down list. To use a board specific AMX clock driver or to define your own **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 code fragment created by the AMX Configuration Builder in your Target Configuration Module. The entry point to the ISP root is declared with a public symbol defined with the name you provide. The name must be unique and must conform to the symbol naming conventions of your assembler.

## Interrupt Handler

Enter the name of your device Interrupt Handler which will complete service of the device. This is the name of the procedure which will be called from the ISP root when the interrupt occurs. The Interrupt Handler can use AMX services. Your Interrupt Handler must be coded as described in Chapter 3.4.

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

## Handler Language

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

## ISP Parameter

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

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

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

## Interrupt Source

You must specify the interrupt source which the ISP will service. From the pull down list, select the interrupt source (Hw\_Int0 to Hw\_Int5) which generates the interrupt which your ISP will service. Requests Hw\_Int0 to Hw\_Int5 correlate directly to the six sources of internal or external hardware interrupt defined by the MIPS32 architecture.

To identify a software interrupt or some other exception as an interrupt source, select Custom from the pull down list. Be sure to edit the interrupt disable mask to meet your custom requirements.

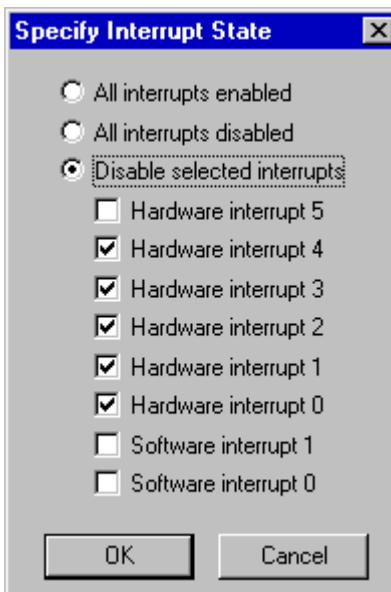
## Interrupt Disable Mask

AMX services interrupt requests in the order of priority defined in Figure 3.3-1. The ISP root adjusts the AMX interrupt disable mask to ensure that the interrupt is serviced at the correct AMX priority level. When you identify the interrupt source, the disable mask field is automatically updated to the value required by the AMX hardware interrupt prioritization scheme. Only the six internal and external hardware interrupt sources are affected; software interrupts 0 and 1 are left enabled.

The disable mask identifies the hardware and software interrupts which must be disabled when servicing the hardware interrupt exception. If necessary, you can edit the mask value directly or use the [...] button to access the interrupt state dialog window shown below. Be sure to disable the hardware interrupt being serviced and all other interrupt exceptions deemed to be of lower priority.

For example, if software interrupt 1 is used to simulate a device interrupt, you must edit the mask for each of your higher priority ISPs to disable software interrupt 1 by ORing the value `0x02` into the mask for your ISP.

When defining an ISP for a device which interrupts via a multiplexed hardware interrupt, the disable mask is not editable. The disable mask which will be used when servicing the device is provided in the definition of the multiplexed hardware interrupt exception.



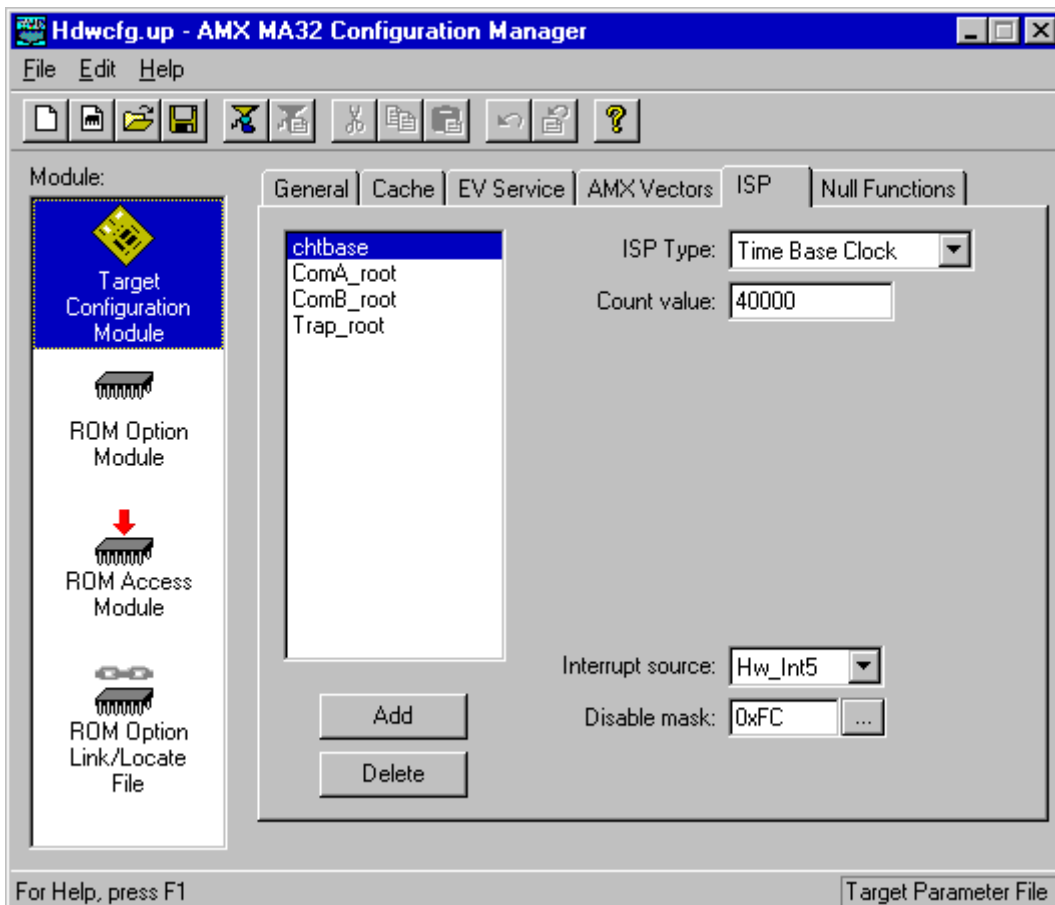
## Using the MIPS32 Count/Compare Interrupt as the AMX Clock

In order to use the AMX time base clock driver as the source of your AMX clock tick, you must provide an ISP definition which declares the ISP to be a Time Base Clock. Then follow the procedures described in Chapter 5.3.1 to include the AMX time base clock driver as part of your AMX application.

Enter the **Count value** required to generate timer count comparison 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 counter frequency can also be dynamically adjusted at run time. To do so, configure the count value to be 0. Your *main()* program must then install the real count value into *long* variable *cj\_tbcount* prior to launching AMX. Thereafter, any change which you make to the value of variable *cj\_tbcount* will take effect at the next count comparison interrupt.

Identify the hardware **Interrupt source** to which the count/compare match interrupt has been assigned. For most MIPS32 processors, the interrupt is generated when the match logic sets *IP[7]=1* in the cause register. Hence, in most cases, simply select *Hw\_Int5* from the pull down list. If necessary, edit the AMX interrupt disable mask to meet your special needs.



## Exception Service Procedure as a Nonconforming ISP

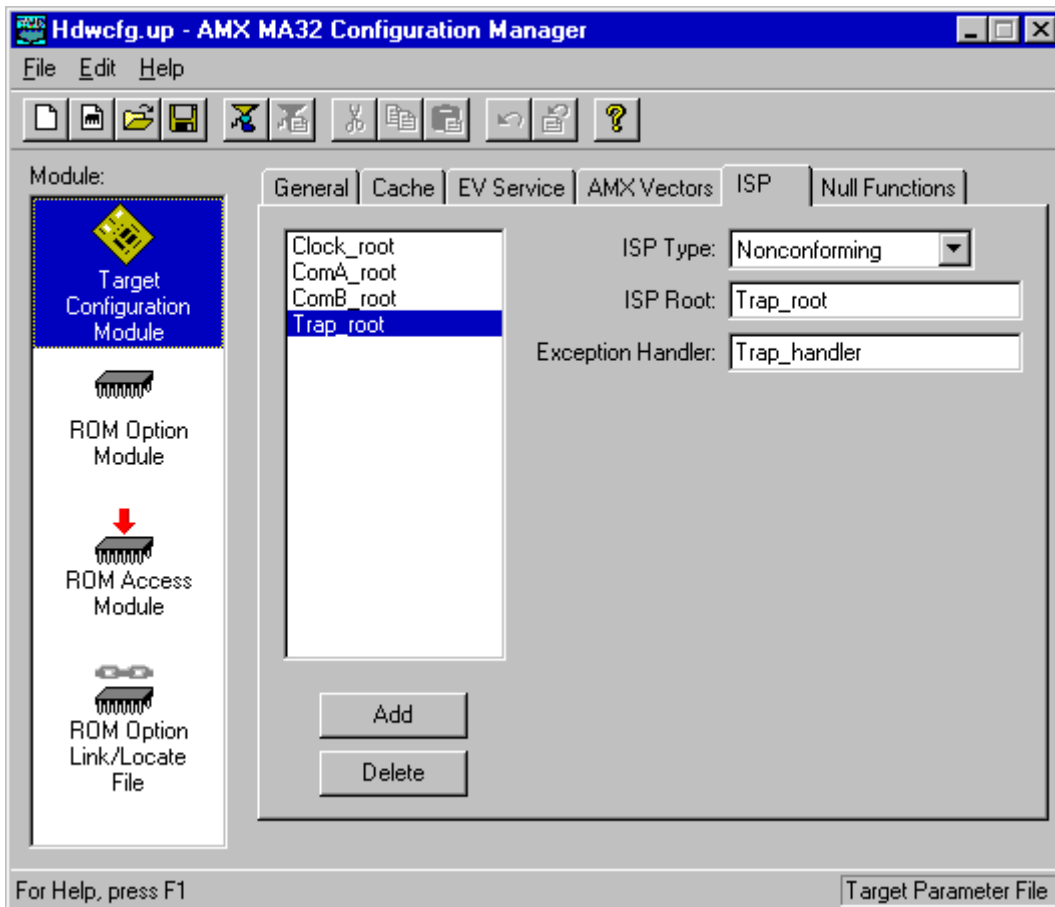
Writing an exception service procedure to handle a non-interrupt exception is rarely easy, especially if the handler must be coded using assembly language. AMX offers a solution.

An exception service procedure can be written in C if you use a nonconforming ISP root to service the exception. The ISP root will be constructed for you by the AMX Configuration Builder from the ISP description 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

To create a **nonconforming AMX ISP** to service a nonconforming interrupt or a non-interrupt exception, choose Nonconforming from the pull down list. ISPs of this type are described in Chapter 3.5

## ISP Root

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

The ISP root is a code fragment created by the AMX Configuration Builder in your Target Configuration Module. The entry point to the ISP root 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.

## Exception Handler

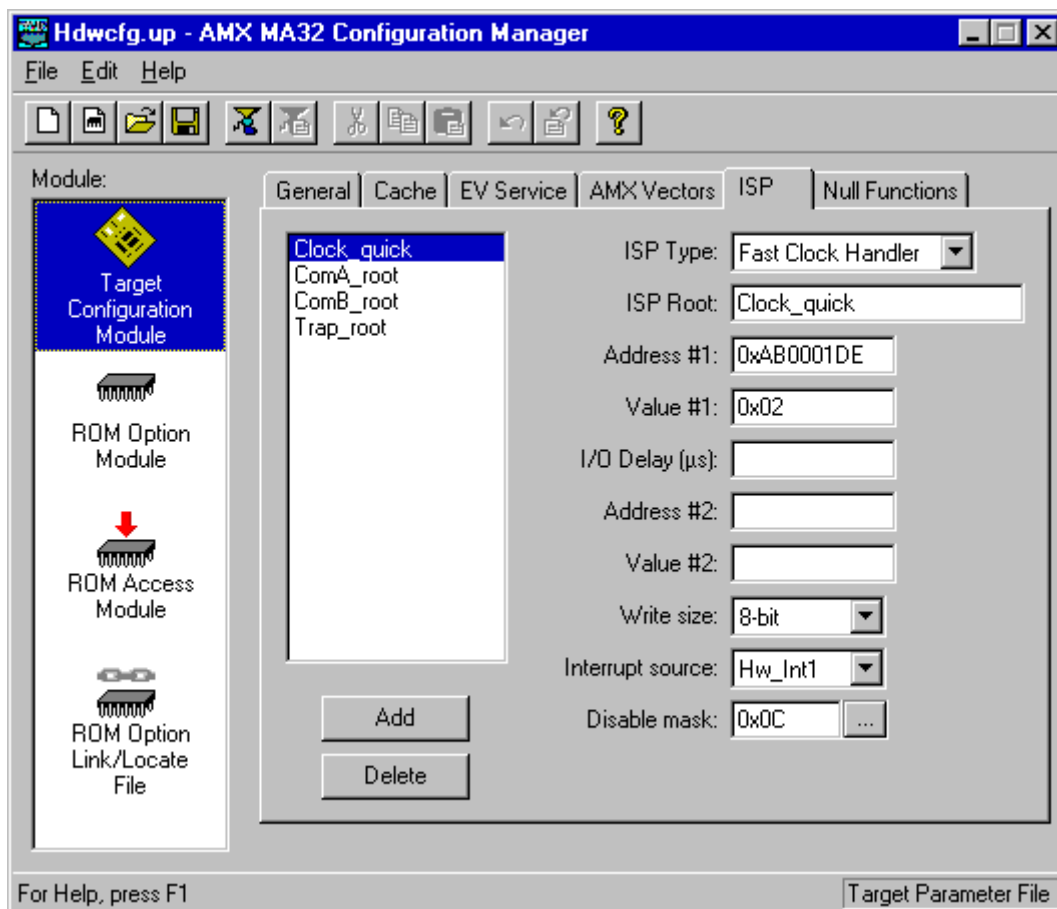
Enter the name of your C function which will complete service of the exception. This is the name of the procedure which will be called from the ISP root when the exception occurs. The function must NOT use AMX services. Your function must be coded in C as described in Chapter 3.5.

## 4.4 Defining a Fast Clock ISP

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

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

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



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

## Clock Service

Your clock device will be serviced as follows:

- Write Value #1 to the device port at memory Address #1.
- Delay for the number of  $\mu\text{s}$  defined as I/O Delay ( $\mu\text{s}$ ).
- Write Value #2 to the device port at memory Address #2.

## Address and Value

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

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

## I/O Delay ( $\mu\text{s}$ )

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  $n \mu\text{s}$  between the I/O device references by entering a non-zero value into this field.

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

## Write Size

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

## Interrupt Source and Disable Mask

You must specify the interrupt source which generates the clock interrupt. From the pull down list, select the interrupt source (Hw\_Int0 to Hw\_Int5). If necessary, you can customize the interrupt source and/or disable mask as described in Chapter 4.3 for other conforming ISPs.

## 4.5 Null Functions

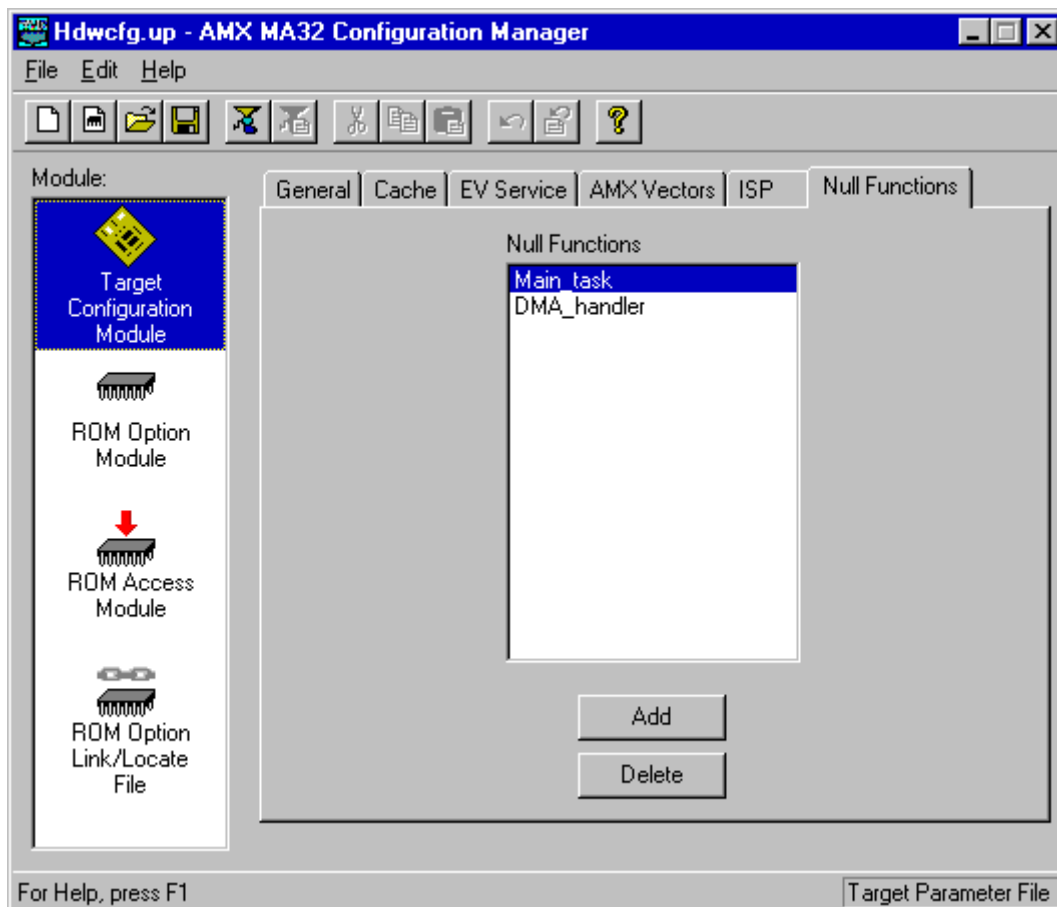
Occasionally, while developing an AMX application, it can be very convenient to be able to create software functions to satisfy your program link requirements without having to create the final version of these functions. For example, if your AMX System Configuration Module references a Restart Procedure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



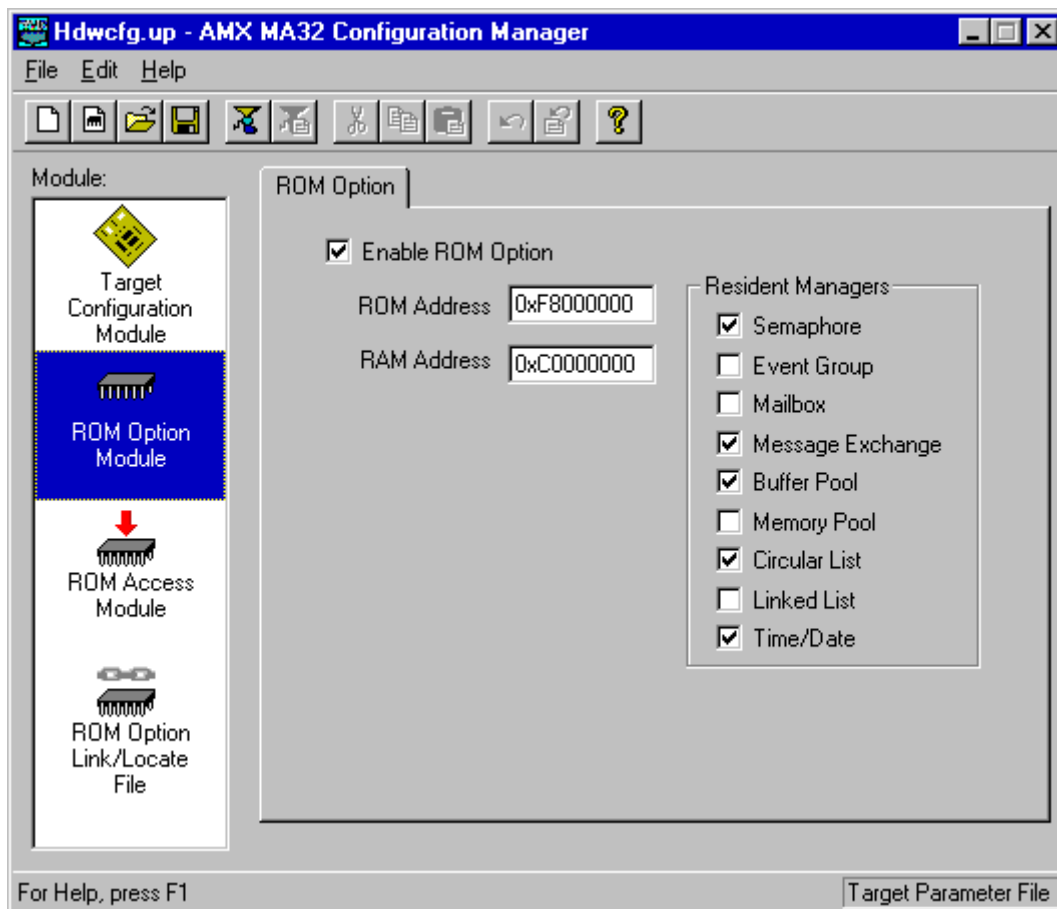
## 4.6 ROM Option Parameters

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.

## 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 MIPS32 processor will interrupt via the general exception vector. 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 and an Interrupt Handler. The ISP root is called by the AMX exception handler in response to the clock interrupt request. The ISP root calls the clock Interrupt Handler to dismiss the clock interrupt request. Your clock ISP must be defined as a conforming ISP of type Clock Handler as described in Chapter 4.3.

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

You may choose to use the AMX time base clock driver which has an ISP root but no Interrupt Handler. The ISP root simply dismisses the clock interrupt request. Such a clock ISP is defined to be an ISP of type Time Base Clock as described in Chapter 4.3.

It is the ISP root which informs AMX that a hardware clock tick has occurred. When you define your clock ISP, your definition of the ISP as a Clock Handler (or Fast Clock Handler or Time Base Clock) 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 `0xAE000040`.

The clock generates hardware interrupt 2.

The clock is therefore assigned to vector `CJ_PRVNIN2` in the AMX Vector Table.

The clock interrupt is dismissed by writing bit pattern `0x08` to the 32-bit clock register at its base address plus 4.

An assembly language conforming clock Interrupt Handler for such a device could be coded as follows.

```
.globl clockih
clockih:
    /* receives v0 = ISP root parameter = A(clock base) */
    ori    v1,zero,8      /* v1 = bit pattern = 8 */
    jr     ra              /* Return */
    sw     v1,4(v0)       /* Delay: dismiss interrupt */
```

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:	<code>clockroot</code>
Interrupt Handler:	<code>clockih</code>
Handler Language:	Assembly
Parameter Type:	Value
Parameter:	<code>0xAE000040</code>
Interrupt source:	Hw_Int2
Disable mask:	<code>0x1C</code>

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

ISP Type:	Fast Clock Handler
ISP Root:	<code>clockroot</code>
Address #1:	<code>0xAE000044</code>
Value #1:	<code>0x08</code>
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	32-bit
Interrupt source:	Hw_Int2
Disable mask:	<code>0x1C</code>

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 */
    cjsivtwr(CJ_PrvNIN2, (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 *chxxxxxxxx*.

### 5.3.1 Time Base Clock Driver

Most MIPS32 processors include a count register in coprocessor 0 (CP0) which is automatically incremented at a fixed frequency, usually determined by a subdivision of the bus frequency. The 32-bit timer count wraps from *0xFFFFFFFF* to *0*. A separate 32-bit compare register in CP0 acts as an interrupt source, generating a general exception interrupt whenever the count register value matches the compare register value.

For most MIPS32 processors, a count/compare match generates hardware interrupt 5 by setting *IP[7]* in the CP0 cause register. The interrupt is dismissed by writing a new value to the compare register. Some, but not all, processors permit the comparison interrupt to be disabled so that hardware interrupt 5 can be used for other purposes. The MIPS32 architecture also permits the comparison interrupt to be assigned to a different hardware interrupt number.

The AMX time base clock driver is ready for use with any MIPS32 processor equipped with the necessary CP0 count/compare timer logic. **Source code** for this AMX clock driver is generated in the Target Configuration Module produced by the AMX Configuration Builder.

To use the AMX time base clock driver, you must define an AMX clock ISP as described in Chapter 4.2. Use the following parameters in your ISP definition.

Purpose:	Time Base Clock
Count value:	32-bit count value which establishes the clock frequency
Interrupt source:	Hardware interrupt number (usually Hw_Int5)
Disable mask:	Interrupt disable mask (usually <i>0xFC</i> )

The **Count value** is a 32-bit positive value used to define the AMX clock interrupt frequency. For most MIPS32 processors, the count timer frequency is 1/2 of the instruction clock frequency. The count value *TCOUNT* can be computed using the following formula:

$$TCOUNT = p * (f/2)$$

*p* = required clock period measured in microseconds.

*f* = processor instruction clock frequency expressed in MHz.

Example:

*p* = 1000  $\mu$ s (for a 1 KHz count/compare interrupt)

*f* = 40 MHz

$$TCOUNT = 1000 * (40/2) = 20000$$

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

You must also identify the hardware **Interrupt source** to which the count/compare match interrupt has been assigned. For most MIPS32 processors, the interrupt is generated when the match logic sets `IP[7]=1` in the cause register. Hence, in most cases, simply select `Hw_Int5` from the pull down list. If necessary, edit the AMX interrupt **Disable mask** to meet your special needs.

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

When AMX is launched, a very long compare value is loaded into the compare register to prevent interrupts during the launch. The clock initialization procedure `chclockinit` will then install the pointer to the clock ISP root `chtbclock` into the specified vector in the AMX Vector Table and initialize the compare logic to generate interrupts at the specified frequency.

Thereafter, as comparison interrupts occur, the AMX exception handler branches via the entry in the AMX Vector Table to the ISP root `chtbclock`. The ISP root clears the interrupt by updating the compare register and then declares 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 operating 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 reset the clock interrupt mask bit in the `IM` field of the status register to prevent clock interrupts until the original general exception handler and status register content have been restored.

### 5.3.2 8254 Clock Driver

The AMX clock driver for the Intel 8254 counter/timer chip (or equivalent) is ready for use on the MIPS Malta 4Kc Development 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 *CH8254T.C*.

The 8254 timer generates IRQ0 on the master 8259 interrupt controller forcing an interrupt request via the MIPS32 hardware interrupt 0. Board support module *MALTA4KC.S* provides interrupt and clock support services used by this clock driver.

You must compile clock source module *CH8254T.C*, assemble board support module *MALTA4KC.S* and link the resulting object modules with the rest of your AMX application.

To use the AMX 8254 clock driver on the Malta 4Kc Development Board, you must define hardware interrupt 0 to be a multiplexed interrupt exception. Use the following parameters in your definition of the AMX vector for hardware interrupt 0.

Purpose:	Multiplexed
Interrupt Identification Procedure:	<i>ch8259vd</i>
# of devices:	16
First vector #:	0
Runtime vector checking:	unchecked
Prioritized/nested by software:	checked
IIP storage (bytes):	4
Disable mask:	0x04

Interrupt Identification Procedure *ch8259vd()* in board support module *MALTA4KC.S* will identify the interrupt source as device number 0 to 15. A block of 16 AMX vectors are reserved starting at AMX vector number *CJ\_PRVNMUX+0*. The first 8 vectors are used for devices connected to the master 8259 interrupt controller. The next 8 vectors are used for devices connected to the slave 8259 interrupt controller. Since the 8254 timer generates master IRQ0, it is device number 0 and is serviced via AMX vector *CJ\_PRVNMUX+0+0*.

To use the AMX 8254 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:	<i>ch8254clk</i>
Interrupt Handler:	<i>ch8254ih</i>
Handler Language:	C
Parameter Type:	(none)
Interrupt source:	Hw_Int0 (multiplexed)

The resulting Target Configuration Module will include a clock ISP root named *ch8254clk*. The clock driver's initialization procedure will install the pointer to this clock ISP root into vector number *CJ\_PRVNMUX+0+0* in the AMX Vector Table. When the interrupt is serviced, the ISP root calls the Interrupt Handler *ch8254ih* in board support module *MALTA4KC.S* to dismiss the timer interrupt. The timer automatically restarts with the required period.

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 or change the timer frequency, 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 AMX vector number, you must edit the definitions in source file *CH8254T.C* and recompile the module. You must also edit the definitions in board support module *MALTA4KC.S* and reassemble the module. Edit instructions are included in the files.

The board support module *MALTA4KC.S* includes a board initialization procedure *chbrdinit* which can be modified to initialize the interrupt controller for your particular board. It is recommended that *chbrdinit* be called from your *main* program prior to launching AMX. Alternatively, include *chbrdinit* as the first procedure in your list of Restart Procedures.

### Dedicated Clock Interrupt

If you wish to use the 8254 clock driver in a system which directly connects the Intel 8254 counter/timer chip to a specific MIPS32 hardware interrupt, you must edit the definitions in source file *CH8254T.C* and board support module *MALTA4KC.S*. Change the vector definition to specify the AMX vector number for the hardware interrupt to which the timer is connected. Your AMX Target Parameter File must declare that the AMX vector is a User exception, not Multiplexed.

Revise functions *chclken()* and *chclkdis()* to enable and disable the generation of interrupts by the counter/timer chip. Alter function *ch8254ih()* to dismiss the 8254 clock interrupt request. Alternatively, you may be able to eliminate Interrupt Handler function *ch8254ih()* and use a Fast Clock Handler instead (see Chapter 5.2).

You must compile clock source module *CH8254T.C*, assemble board support module *MALTA4KC.S* and link the resulting object modules with the rest of your AMX application.

## 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<sup>®</sup> utility provided with AMX.

```
; AMX Target Parameter File
:
...LAUNCH          PERM
...HDW             PROC ,DMASK ,SRMASK ,CACHE
...CACHE          FNCACHE ,ICSIZE ,ICPARAM ,DCSIZE ,DCPARAM
...DELAY          CPUFREQ
...VUSER          VUSER0 ,VUSER1
...VPLUG          VPLUG0 ,VPLUG1
...VPASS          VPASS0 ,VPASS1
...VECTOR         VLOC
...VDEFN          VTYPE ,VATTR ,BEV0ADR ,BEV1ADR ,PTPARAM
...ISPMUX         VIDPROC ,VNBASE ,VNCOUNT ,VNCHECK ,VNEST ,INTSRC ,DMASK
;
;               ISP definitions (one line for each conforming ISP)
...ISPA          ISPROOT ,HANDLER ,VNUM ,PARAM ,PARTYPE ,INTSRC ,DMASK
...ISPC          ISPROOT ,HANDLER ,VNUM ,PARAM ,PARTYPE ,INTSRC ,DMASK
;
;               Nonconforming ISP definitions (one line for each exception)
...NCISP         ISPROOT ,FUNCTION ,VNUM
;
;               CP0 time base clock ISP (no user code required)
...CLKTBASE      TCOUNT ,INTSRC ,DMASK
;
;               Conforming fast clock ISP (no user code required)
...CLKFAST       CLKROOT ,CLKADR ,CLKCMD ,CLKADR2 ,CLKCMD2 ,IODELAY ,VNUM ,
                INTSRC ,DMASK
...CLKFAST16     parameters are same as ...CLKFAST
...CLKFAST32     parameters are same as ...CLKFAST
;
;               or conforming clock ISP (coded in assembly language)
...CLKA          CLKROOT ,CLKHAND ,VNUM ,PARAM ,PARTYPE ,INTSRC ,DMASK
;
;               or conforming clock ISP (coded in C)
...CLKC          CLKROOT ,CLKHAND ,VNUM ,PARAM ,PARTYPE ,INTSRC ,DMASK
;
;               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 MA32 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 , DMASK , SRMASK , CACHE

          PROC      Processor identifier
          DMASK     0xMM00 = AMX task level interrupt disable mask
          SRMASK    0xSS00 = Initial interrupt inhibit mask
          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:

<i>MIPS32</i> ,	{MIPS32 architecture compliant}
<i>4KC</i> , <i>4KM</i> , <i>4KP</i> ,	{MIPS 4K cores or equivalent}
<i>5K</i>	{MIPS 5K core or equivalent}

The *DMASK* parameter establishes the value of the AMX interrupt disable mask when tasks are executing. The AMX interrupt priority mechanism is described in Chapter 3.3. For most systems, this parameter is 0 since all six internal and external hardware interrupts and both software interrupts are normally enabled when tasks are running.

The *SRMASK* parameter can be used to adjust the initial state of the *IM* field in the processor status register at launch time. When AMX is launched, AMX disables the interrupt system by resetting bit *IE* in the status register to 0. AMX then sets the status register interrupt mask (*IM*) to enable all interrupt sources **except** those specified by *SRMASK*. AMX then selectively disables the interrupts specified by *DMASK*. Prior to calling your Restart Procedures, AMX enables the interrupt system by setting bit *IE* in the status register to 1.

The *CACHE* parameter can be used to instruct AMX to enable the processor 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       MIPS32 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.

## AMX Vector Usage

The Target Parameter File includes a set of definitions which indicate the manner in which AMX must service each of the exceptions which are dispatched via the AMX Vector Table (see Figure 3.2-1).

```
...VUSER          VUSER0 ,VUSER1
...VPLUG          VPLUG0 ,VPLUG1
...VPASS          VPASS0 ,VPASS1
```

Directive `...VUSER` identifies AMX vectors which will be dispatched to user handlers provided by your application.

Directive `...VPLUG` identifies AMX vectors which will be plugged by AMX.

Directive `...VPASS` identifies AMX vectors which will be passed by AMX through to the original exception handler which was in effect prior to the launch of AMX. AMX vectors which are declared as pass through will be treated as fatal if the processor exception vector which handles the exception is not also defined as pass through.

If an AMX vector is identified by multiple directives, the vector will be treated as user, plugged or pass through, in that order of precedence. All AMX vectors which are not specified by one of these directives will be treated as fatal by AMX.

Parameters `Vxxxxx0` and `Vxxxxx1` are unsigned 32-bit hexadecimal values which form a 64-bit mask value. Each bit in the mask identifies one of 64 AMX vectors. Bit 0 of `Vxxxxx0` corresponds to AMX vector number 0. Bit 31 of `Vxxxxx1` corresponds to AMX vector number 63. AMX vectors are identified by mask bits which are set to 1.

## Exception Vector Addresses

The Target Parameter File includes a definition which permits AMX to derive or identify the memory address of each of the processor exception vectors which AMX must support. The vector addresses can be fixed or dynamically selected according to the setting of the *BEV* bit in the processor status register at the time AMX is launched.

```
...VECTOR          VLOC
...VDEFN           VTYPE, VATTR, BEV0ADR, BEV1ADR, PTPARAM
```

*VLOC*            *OxNM* = Exception vector location specification  
*M=0* if normal (*BEV=0*) vector addresses are to be assumed  
*M=1* if bootstrap (*BEV=1*) vector addresses are to be assumed  
*M=2* if vector addresses are to be determined by the setting of *BEV*  
*N=0/1* if vector addresses are adjustable by user (editable)

For each exception vector which AMX is to service, there must be a *VDEFN* directive to establish the attributes and location of the exception. Omit the *VDEFN* directive for a particular exception if AMX is not to service that exception.

The *VTYPE* parameter identifies the exception vector.

*VTYPE*            Exception vector type:  
0    General exception vector  
1    Special interrupt exception vector  
2    TLB refill exception vector  
3    Cache error exception vector  
4    Debug exception vector  
5    Reset exception vector (defined but not supported)

The *VATTR* parameter identifies the exception vector attributes.

*VATTR*            *OxON* = Exception vector attributes  
*N[0]=0/1* if vector memory cannot/can be modified by AMX  
*N[1]=0/1* if exception cannot/can be passed through by AMX  
                  to the original exception handler  
*N[2]=0/1* to select the intercept/branch pass through strategy

Parameters *BEV0ADR* and *BEV1ADR* specify the address in memory at which the exception vector is located. Address *BEV0ADR* is used if addressing is dictated by a *BEV* value of 0. Address *BEV1ADR* is used if addressing is dictated by a *BEV* value of 1.

Parameter *PTPARAM* is the pass through parameter. If the *VATTR* parameter indicates that pass through support is enabled using the intercept strategy, then parameter *PTPARAM* is the number of bytes at the vector location which must be saved and executed by AMX to pass on an exception. If the *VATTR* parameter indicates that pass through support is enabled using the branch strategy, then parameter *PTPARAM* is the memory address to which AMX must branch to pass on an exception.

## Multiplexed Hardware Interrupt Exception

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

```
...ISP_MUX          VIDPROC, VNBASE, VNCOUNT, VNCHECK, VNEST, INTSRC, DMASK

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 multiplexed hardware interrupt exception
VNCHECK            0 if run-time vector number checking is disabled
                   1 if run-time vector number checking is enabled
VNEST              0 if interrupt nesting is not allowed
                   Number of bytes of storage needed to support interrupt nesting
INTSRC             Interrupt source (hardware interrupt number)
DMASK              Interrupt disable mask
```

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.

A block of  $n$  vectors in the AMX Vector Table is reserved for every multiplexed interrupt. These reserved blocks start at AMX vector number *CJ\_PRVNMUX*. Parameter *VNBASE* defines the offset from *CJ\_PRVNMUX* of the base of the block of vectors reserved by you for the devices attached to a particular interrupt exception. The interrupt identification number provided by your Interrupt Identification Procedure is added to *CJ\_PRVNMUX+VNBASE* to derive the AMX vector number for the interrupting device.

Parameter *VNCOUNT* defines the number of AMX vectors to be allocated to the particular interrupt exception. The devices are numbered from 0 to  $VNCOUNT-1$ . The AMX vector number for a device is derived by adding its device number to the base AMX vector number *CJ\_PRVNMUX+VNBASE*. 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 multiplexed interrupt exception handler can check that the identified device number lies within the range 0 to  $VNCOUNT-1$ . If it does not, AMX calls the Fatal Exception Handler with AMX vector number *CJ\_PRVNMUX+i* as a parameter indicating that an unidentified interrupt exception occurred via hardware interrupt number  $i$ .

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 your Interrupt Identification Procedure does not support nested interrupts, set parameter *VNEST* to 0. Otherwise, set *VNEST* to the number of bytes of storage required by your IIP to support nesting of interrupts. Instructions for creating such an IIP are provided in Appendix E. The AMX multiplexed interrupt exception handler will allocate *VNEST* bytes of storage on the AMX Interrupt Stack for use by your IIP. The numeric value must be a multiple of 4 and must be expressed in a form acceptable to your assembler. If your IIP does not require storage to track the nesting of interrupts, set *VNEST* to 4 to minimize the allocated storage while still indicating that nesting is supported.

Parameter *INTSRC* defines the hardware interrupt number which generates the multiplexed interrupt exception. *INTSRC* values of 0 to 5 correspond to internal or external hardware interrupts Hw\_Int0 to Hw\_Int5 which are identified by *IP[2]* to *IP[7]* respectively in the processor cause register.

The *DMASK* parameter establishes the value of the AMX interrupt disable mask when your IIP and Interrupt Handler is executed, thereby establishing the effective interrupt priority. The AMX interrupt priority mechanism is described in Chapter 3.3. For most systems, this parameter will assume one of the values defined in Figure 3.3-1. The *DMASK* value **must** inhibit the multiplexed hardware interrupt exception being serviced.

If interrupt nesting is supported, the AMX multiplexed interrupt exception handler will manipulate the AMX disable mask to permit nesting of interrupts as described in Appendix E.

## 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 one of the following directives. The ISP root is declared using `...ISP_C` if its Interrupt Handler is coded in C or `...ISP_A` if its Interrupt Handler is coded in assembly language.

```
...ISP_A      ISPROOT, HANDLER, VNUM, PARAM, PARTYPE, INTSRC, DMASK
...ISP_C      ISPROOT, HANDLER, VNUM, PARAM, PARTYPE, INTSRC, DMASK
```

<i>ISPROOT</i>	Name of the ISP root entry point
<i>HANDLER</i>	Name of the public device Interrupt Handler
<i>VNUM</i>	AMX vector number assigned to the device
<i>PARAM</i>	Parameter for use by the Interrupt Handler
<i>PARTYPE</i>	Parameter <i>PARAM</i> type
<i>INTSRC</i>	Interrupt source (hardware interrupt number)
<i>DMASK</i>	Interrupt disable mask

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

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

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

*VNUM* defines the AMX vector number which you have assigned to the device. *VNUM* must be a value from 0 to *NVEC*-1, where *NVEC* is the total number of vectors in the AMX Vector Table. If you have no multiplexed interrupt exceptions, then *NVEC* will match the definition of *CJ\_PRVNMUX*. If one or more multiplexed interrupt exceptions are present, *NVEC* will be *CJ\_PRVNMUX* plus the maximum value of *VNBASE*+*VNCOUNT* specified by your `...ISPMUX` directives.

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.

Parameter *INTSRC* defines the hardware interrupt number which generates the interrupt exception. *INTSRC* values of 0 to 5 correspond to internal or external hardware interrupts Hw\_Int0 to Hw\_Int5 which are identified by *IP[2]* to *IP[7]* respectively in the processor cause register.

An *INTSRC* value of -1 can be used to represent some other source of interrupt such as a software interrupt or a processor exception used to simulate an interrupt.

When defining the ISP for a device which is serviced via a multiplexed hardware interrupt, parameter *INTSRC* must identify the multiplexed hardware interrupt. Hence, all of the ISP descriptions for a collection of multiplexed devices will have the same value for parameter *INTSRC*.

The *DMASK* parameter establishes the value of the AMX interrupt disable mask when your Interrupt Handler is executing, thereby establishing the effective interrupt priority. The AMX interrupt priority mechanism is described in Chapter 3.3. For most systems, this parameter will assume one of the values defined in Figure 3.3-1.

When defining the ISP for a device which is serviced via a multiplexed hardware interrupt, parameter *DMASK* must be 1. The actual AMX interrupt disable mask which will be used when servicing the device is provided in the *...ISPMUX* directive which describes the multiplexed hardware interrupt exception.

### 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. Note that an exception handler for a non-interrupt exception is also considered to be a nonconforming ISP. The ISP root definition is provided using the following directive. Note that the nonconforming ISP function must be coded in C.

```
...NCISP          ISPROOT,FUNCTION,VNUM

    ISPROOT      Name of the ISP root entry point
    FUNCTION     Name of the public C function to be called from the ISP root
    VNUM        AMX vector number assigned to the exception
```

*VNUM* defines the AMX vector number through which the nonconforming interrupt or exception is serviced. *VNUM* must be a value from 0 to *NVEC-1*, where *NVEC* is the total number of vectors in the AMX Vector Table. *VNUM* is subject to the same restrictions described for the *VNUM* parameter of a conforming ISP.

## AMX Clock Handler Declaration

The Target Parameter File must include a definition of an ISP root for your AMX clock handler. The clock ISP root definition must be provided using one of the following directives. The clock ISP root is declared using `...CLKC` if its Interrupt Handler is coded in C or `...CLKA` if its Interrupt Handler is coded in assembly language. The clock ISP root can be declared using `...CLKFAST` if an Interrupt Handler is not required to service the clock. Use directive `...CLKTBASE` if the MIPS32 internal CP0 count register is to be used as the time base for your AMX clock.

```
...CLKC          CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE,INTSRC,DMASK
...CLKA          CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE,INTSRC,DMASK
...CLKFAST       CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM,
                  INTSRC,DMASK
...CLKFAST16     parameters are same as ...CLKFAST
...CLKFAST32     parameters are same as ...CLKFAST
...CLKTBASE      TCOUNT,INTSRC,DMASK
```

<i>CLKROOT</i>	Name of the clock ISP root entry point
<i>CLKHAND</i>	Name of the public clock device Interrupt Handler
<i>VNUM</i>	AMX vector number assigned to the clock device
<i>PARAM</i>	Parameter for use by the Interrupt Handler
<i>PARTYPE</i>	Parameter <i>PARAM</i> type
<i>INTSRC</i>	Interrupt source (hardware interrupt number)
<i>DMASK</i>	Interrupt disable mask

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

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

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

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

### Note

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

Parameter *INTSRC* defines the hardware interrupt number which the clock uses to generate the interrupt exception. *INTSRC* values of 0 to 5 correspond to internal or external hardware interrupts Hw\_Int0 to Hw\_Int5 which are identified by *IP[2]* to *IP[7]* respectively in the processor cause register.

An *INTSRC* value of -1 can be used to represent some other source of clock interrupt such as a software interrupt or a processor exception used to simulate a clock interrupt.

The *DMASK* parameter establishes the value of the AMX interrupt disable mask when your clock Interrupt Handler is executing, thereby establishing the effective interrupt priority. The AMX interrupt priority mechanism is described in Chapter 3.3. For most systems, this parameter will assume one of the values defined in Figure 3.3-1.

If your clock device is serviced via a multiplexed hardware interrupt, parameter *INTSRC* must identify the multiplexed hardware interrupt and parameter *DMASK* must be 1. The actual AMX interrupt disable mask which will be used when servicing the device is provided in the *...ISPMUX* directive which describes the multiplexed hardware interrupt exception.

## Fast Clock ISP

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

```

...CLKFAST          CLKROOT , CLKADR , CLKCMD , CLKADR2 , CLKCMD2 , IODELAY , VNUM ,
                    INTSRC , DMASK
...CLKFAST16       parameters are same as ...CLKFAST
...CLKFAST32       parameters are same as ...CLKFAST

```

<i>CLKROOT</i>	Name of the clock ISP root entry point
<i>CLKADR</i>	32-bit numeric device memory address
<i>CLKCMD</i>	8-bit numeric command
<i>CLKADR2</i>	32-bit numeric secondary device memory address
<i>CLKCMD2</i>	8-bit numeric secondary command
<i>IODELAY</i>	Delay ( $\mu$ s) required between I/O commands
<i>VNUM</i>	AMX vector number assigned to the clock device
<i>INTSRC</i>	Interrupt source (hardware interrupt number)
<i>DMASK</i>	Interrupt disable mask

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.

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).

Parameters *VNUM*, *INTSRC* and *DMASK* are as previously described for general clock handlers. If parameter *VNUM* is omitted, then a value of *-1* is assumed for *VNUM*.

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

### Time Base Clock Driver

Most MIPS32 processors include a count register in coprocessor 0 (CP0) which is automatically incremented at a fixed frequency, usually determined by a subdivision of the bus frequency. The 32-bit timer count wraps from *0xFFFFFFFF* to *0*. A separate 32-bit compare register in CP0 acts as an interrupt source, generating a general exception interrupt whenever the count register value matches the compare register value.

To use the CP0 count/compare registers as the source of an AMX clock interrupt, include the following directive in your Target Parameter File.

```
...CLKTBASE          TCOUNT,INTSRC,DMASK
```

Parameter *TCOUNT* is the 32-bit positive value used to define the AMX clock interrupt frequency (See Chapter 5.3.1).

Parameter *INTSRC* defines the hardware interrupt number which the compare logic uses to generate the interrupt exception. For most MIPS32 processors, a count/compare match generates an interrupt by setting *IP[7]* in the CP0 cause register. Hence, for most processors, set *INTSRC* to 5. If your MIPS processor uses a different hardware interrupt number, adapt the value for *INTSRC* accordingly. *INTSRC* must be a value in the range 0 to 5, corresponding to internal or external hardware interrupts Hw\_Int0 to Hw\_Int5 which are identified by *IP[2]* to *IP[7]* respectively in the processor cause register.

The *DMASK* parameter establishes the value of the AMX interrupt disable mask when the time base clock Interrupt Handler is executing, thereby establishing the effective clock interrupt priority. For most systems, this parameter will assume one of the values defined in Figure 3.3-1. However, the value can be adapted to meet your requirements.

## 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
<i>C</i>	<i>0xABCDEF01</i>	<i>ABCDEF01</i>
<i>Assembler (Intel)</i>	<i>0ABCDEF01H</i>	<i>ABCDEF01</i>
<i>Assembler (Motorola)</i>	<i>\$ABCDEF01</i>	<i>ABCDEF01</i>

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 MA32 Target Configuration Template File *CJ442HDW.CT* and a Target Parameter File called *HDWCFG.UP* is as follows:

```
CJ442CG HDWCFG.UP CJ442HDW.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.

## Appendix B. AMX MA32 Service Procedures

### B.1 Summary of Services

AMX MA32 provides a collection of target dependent AMX service procedures for use with the MIPS32 processor and compatibles and the C compilers which support them. These procedures are summarized below.

#### Interrupt Control (class *ksi*)

<i>cjksitrap</i>	Install a task trap handler
<i>cjksivtp</i>	Fetch pointer to the AMX Vector Table
<i>cjksivtrd</i>	Read an entry from the AMX Vector Table
<i>cjksivtwr</i>	Write an entry into the AMX Vector Table
<i>cjksivtx</i>	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.

<i>cjcfccsetup</i>	Setup C environment
<i>cjcfdi</i>	Disable interrupts
<i>cjcf diprev</i>	Disable interrupts; return previous processor status register
<i>cjcf dmget</i>	Read the AMX interrupt disable mask
<i>cjcf ei</i>	Enable interrupts
<i>cjcf flagrd</i>	Read the processor status register
<i>cjcf flagwr</i>	Write to the <i>IE</i> bit in the processor status register
<i>cjcf flagwrx</i>	Write a new value into the processor status register
<i>cjcf hwdelay</i>	Delay <i>n</i> microseconds
<i>cjcf hwbcache</i>	Enable/disable the instruction and data cache
<i>cjcf hwdcache</i>	Enable/disable the data cache
<i>cjcf hwicache</i>	Enable/disable the instruction cache
<i>cjcf hw bflush</i>	Flush the instruction and data cache
<i>cjcf hwdflush</i>	Flush the data cache
<i>cjcf hwiflush</i>	Flush the instruction cache
<i>cjcf in8</i>	Read an 8-bit input port
<i>cjcf in16</i>	Read a 16-bit input port
<i>cjcf in32</i>	Read a 32-bit input port
<i>cjcf jlong</i>	Long jump to a mark set by <i>cjcf jset</i>
<i>cjcf jset</i>	Set a mark for a subsequent long jump by <i>cjcf jlong</i>
<i>cjcf mcopy</i>	Copy a block of memory
<i>cjcf mset</i>	Set (fill) a block of memory
<i>cjcf out8</i>	Write an 8-bit value to an output port
<i>cjcf out16</i>	Write a 16-bit value to an output port
<i>cjcf out32</i>	Write a 32-bit value to an output port
<i>cjcf srget</i>	Read the considered state of the processor status register
<i>cjcf srset</i>	Alter the AMX interrupt disable mask and/or the considered state of the processor status register

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

<i>cjcfstkjmp</i>	Switch stacks and jump to a new procedure
<i>cjcftag</i>	Convert a string to an AMX tag value
<i>cjcfvol8</i>	Read a volatile 8-bit variable
<i>cjcfvol16</i>	Read a volatile 16-bit variable
<i>cjcfvol32</i>	Read a volatile 32-bit variable
<i>cjcfvolpntr</i>	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 structure *cjxregs* which they use is defined in file *CJZZZKT.H*.

<i>cjcfregld</i>	Load MIPS32 general registers from a register array
<i>cjcfregst</i>	Store MIPS32 general registers into a register array
<i>cjcf sint</i>	Generate a software initiated exception

## B.2 Service Procedures

A description of all processor dependent AMX MA32 service procedures is provided in this appendix. The descriptions are ordered alphabetically for easy reference.

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

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

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

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

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

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

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

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

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

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

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

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

- Disabled
- Enabled  
(Not in ISP)
- Restored

**D E R Effect on Interrupts**

- □ □ Untouched
- □ □ Disabled and left disabled upon return
- ■ □ Enabled and left enabled upon return
- ■ □ Disabled and then enabled upon return
- □ ■ Disabled and then, prior to return, restored to the state in effect upon entry to the procedure
- ■ ■ Disabled, possibly briefly enabled and then, prior to return, restored to the state in effect upon entry to the procedure

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

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

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

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

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

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

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

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

**Purpose**      **Setup C Environment****Used by**      ■ Task    ■ ISP    ■ Timer Procedure    ■ Restart Procedure    ■ Exit Procedure**Setup**      Prototype is in file *CJZZZIF.H*.  

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

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



**Purpose**      **Get the Current AMX Interrupt Disable Mask**

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

**Setup**      Prototype is in file *CJZZZIF.H*.  
                  *#include "CJZZZ.H"*  
                  *CJ\_T32U CJ\_CCPP cjcfdmget(void);*

**Description**    AMX maintains a private copy of the considered state of the processor status register and a private mask used to identify interrupts which must be disabled. These private variables are used to define the actual state of the processor status register. Use *cjcfdmget* to determine which hardware interrupts are currently disabled by the AMX interrupt disable mask.

**Interrupts**     Disabled     Enabled     Restored

**Returns**      The current value of the AMX interrupt disable mask.

**Note**          The return value will indicate which, if any, of the hardware and software interrupts are currently disabled by the AMX disable mask. Use *cjsrget* to determine which, if any, of these interrupts would be enabled if they were not disabled by the mask.

**See Also**      *cjcfsrget*, *cjcfsrset*

**cjcfflagrd  
cjcfflagwr  
cjcfflagwrx**

**cjcfflagrd  
cjcfflagwr  
cjcfflagwrx**

**Purpose** Read or Write Processor Status Register

**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_TYFLAGS CJ_CCPP cjcfflagrd(void);  
void CJ_CCPP cjcfflagwr(CJ_TYFLAGS flags);  
void CJ_CCPP cjcfflagwrx(CJ_TYFLAGS flags);
```

**Description** *Cjcfflagrd* returns the actual state of the processor status register.

*Cjcfflagwr* restores only the interrupt enable bit *IE* in the processor status register from parameter *flags*. *Cjcfflagwrx* unconditionally copies the the value *flags* into the processor status register.

Use *cjcfflagrd* to read the state of the processor status register, thereby capturing the current interrupt state. Then use *cjcfdi* to briefly disable all sources of interrupt. Immediately thereafter, use *cjcfflagwr* to restore the state of the interrupt system.

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

**Returns** *Cjcfflagrd* returns the actual state of the processor status register. *Cjcfflagwr* and *cjcfflagwrx* return nothing.

*Cjcfflagwr* updates the *IE* bit in the processor status register to match the value in the corresponding bit of parameter *flags*, thereby enabling or disabling interrupts. The interrupt masks in the processor status register are not altered.

**Restrictions** Procedures *cjcfflagrd* and *cjcfflagwrx* can be used to manipulate the processor status register 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()*. You cannot use procedure *cjcfflagwr* unless AMX is active.

**Warning**

Once AMX has been launched, you must NOT use *cjcfflagwrx* to write to the processor status register. You MUST use procedure *cjcfsrset* to alter the status register content. Failure to observe this restriction may lead to an AMX malfunction.

**See Also** *cjcfdi*, *cjcf diprev*, *cjcf ei*, *cjcf srset*

**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 MIPS32 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. You MUST disable interrupts prior to disabling or enabling the cache. Use *cjcfdiprev* to disable interrupts and *cjcfflagwr* to restore interrupts.

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

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

**cjcfhwbflush**  
**cjcfhwdflush**  
**cjcfhwiflush**

**cjcfhwbflush**  
**cjcfhwdflush**  
**cjcfhwiflush**

**Purpose** Flush (Invalidate) Caches

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

**Setup** Prototypes are in file *CJZZZIF.H*.  

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

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

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

**Interrupts** □ Disabled □ Enabled □ Restored

**Returns** Nothing

*Cjcfhwbflush* flushes both the data cache and instruction cache.

*Cjcfhwdflush* flushes only the data cache.

*Cjcfhwiflush* flushes only the instruction cache.

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

**Note** These procedures can be called even if your Target Parameter File indicates that you are targeting a MIPS32 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 *cjcfddiprev* to disable interrupts and *cjcfflagwr* to restore interrupts.

If you call these procedures from your *main()* program before or after your call to *cjkslaunch()*, you must use *cjcfflagrd* and *cjcfflagwr* to manipulate the status register 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 Interrupt Handlers and Timer Procedures should not use these procedures.

**cjcfm8**  
**cjcfm16**  
**cjcfm32**

**cjcfm8**  
**cjcfm16**  
**cjcfm32**

**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 cjcfm8(void *port);
CJ_T16 CJ_CCPP cjcfm16(void *port);
CJ_T32 CJ_CCPP cjcfm32(void *port);
```

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

**Interrupts** □ Disabled □ Enabled □ Restored

**Returns** *Cjcfm8* returns an 8-bit signed value.  
*Cjcfm16* returns a 16-bit signed value.  
*Cjcfm32* returns a 32-bit signed value.

**Example**

```
#include "CJZZZ.H"

/* Console status register */
#define CONSTAT ((CJ_T8 *)0xAFF8002DL)
/* Console data register */
#define CONDATA ((CJ_T8 *)0xAFF8002FL)

void CJ_CCPP conout(char ch) {
    /* Wait for ready */
    while ( (cjcfm8(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`

**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 *CJZZZF.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));
}
```

## **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 *)0xAFF8002DL)
/* Console data register */
#define CONDATA ((CJ_T8 *)0xAFF8002FL)

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**      **Get the Considered State of the Processor Status Register**

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

**Setup**      Prototype is in file *CJZZZIF.H*.  
                  *#include "CJZZZ.H"*  
                  *CJ\_T32U CJ\_CCPP cjcsrget(void);*

**Description**    AMX maintains a private copy of the considered state of the processor status register and a private mask used to identify interrupts which must be disabled. These private variables are used to define the actual state of the processor status register. Use *cjcsrget* to read the considered state of the processor status register.

**Interrupts**     Disabled     Enabled     Restored

**Returns**      The current value of the AMX copy of the processor status register.

**Note**          The return value reflects the considered state of the status register. The return value will indicate which, if any, of the hardware and software interrupts are considered to be enabled even though the interrupt may actually be disabled in the real processor status register because of the current state of the AMX disable mask.

**See Also**      *cjcfmget, cjcsrset, cjcfflagrd*

**Purpose**      **Set the Processor Status Register**

AMX maintains a private copy of the considered state of the processor status register and a private mask used to identify interrupts which must be disabled. These private variables are used to define the actual state of the processor status register. Use *cjcfsrset* to update the considered state of these private AMX variables and to adjust the actual processor status register accordingly.

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

**Setup**      Prototype is in file *CJZZZIF.H*.  

```
#include "CJZZZ.H"
void CJ_CCPP cjcfsrset(CJ_T32U srmask,
                      CJ_T32U srvalue, CJ_T32U dmvalue);
```

**Description**      *srmask* is the 32-bit mask defining the bits in the AMX status register copy which are to be altered. *srmask* is restricted to bits *0xFFFFFFFF00L*.

*srvalue* is the 32-bit mask defining the values for each of the bits specified by *srmask*. *Srvalue* is restricted to bits *0xFFFFFFFF00L*. If *srvalue* = 1, the AMX status register copy is not altered.

*dmvalue* is the 32-bit mask defining the hardware and software interrupts which are to be disabled. *Dmvalue* is restricted to bits *0x0000FF00L*. If *dmvalue* = 1, the AMX interrupt disable mask is not altered.

**Interrupts**      If *srvalue* is not 1, the AMX status register copy is updated to adjust only those bits indicated by *srmask* to the values indicated by *srvalue*. The *srvalue* bit restrictions are enforced.

If *dmvalue* is not 1, the AMX disable mask is updated to adjust only those bits indicated by *srmask* to the values indicated by *dmvalue*. The *dmvalue* bit restrictions are enforced.

The processor status register is then updated to match the AMX copy with the following exception. The status bits for interrupts which, according to the current AMX interrupt disable mask, are to be disabled are set to 0. The *IE* bit in the processor status register is then set to 1 enabling all interrupts except those explicitly inhibited.

**Returns**      Nothing

**Restrictions**      Once AMX has been launched, you must use *cjcfsrset* to alter bits other than the *IE* bit in the processor status register. Failure to observe this restriction may lead to unexpected and unpredictable faults.

**See Also**      *cjcfdi*, *cjcf diprev*, *cjcfei*, *cjcf dmget*, *cjcf srget*,  
*cjcf flagrd*, *cjcf flagwr*

**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);
}
```

This page left blank intentionally.

**Purpose**      **Install a Task Trap Handler**

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

**Setup**      Prototype is in file *CJZZZIF.H*.  
                  *#include "CJZZZ.H"*  
                  *CJ\_ERRST CJ\_CCPP cjksitrap(int trapid, CJ\_TRAPPROC handler);*

**Description**    *trapid* is the AMX vector number which identifies the particular error trap.

*CJ\_PRVNOVF*      Arithmetic overflow trap

*handler* is a pointer to the task's trap handler for the particular error trap.

**Interrupts**    □ Disabled    □ Enabled    □ Restored

**Returns**      Error status is returned.  
                  *CJ\_EROK*              Call successful.

Errors returned:  
                  *CJ\_ERTKTRAP*      *Trapid* is not a vector number for which task traps are allowed.

## **cjksivtp**

## **cjksivtp**

**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 ckjsivtrd(int vector, CJ\_ISPPROC \*oldproc);*

**Description**      *vector* is the AMX vector number.  
                          *CJ\_PRVNxxxxx*      See definitions in Figure 3.2-1.  
                  *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\_ERRRANGE*      Invalid AMX vector number.

**See Also**      *ckjsivtwr, ckjsivtx*

**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.  
*CJ\_PRVNxxxxx* See definitions in Figure 3.2-1.  
*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\_ERRRANGE* 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.  
*CJ\_PRVNxxxx* See definitions in Figure 3.2-1.  
*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*

This page left blank intentionally.

## Appendix C. AMX MA32 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.

<i>CJ442ROP.LKT</i>	AMX ROM Option toolset dependent Link Specification Template
<i>CJ442ROP.CT</i>	AMX ROM Option Template
<i>CJ442RAC.CT</i>	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 *CJ442ROP.CT* to produce an assembly language ROM Option Module *CJ442ROP.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:

```
CJ442CG HDWCFG.UP CJ442ROP.CT CJ442ROP.S
```

The ROM Option Module *CJ442ROP.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 *CJ442RAC.CT* to produce an assembly language ROM Access Module *CJ442RAC.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:

```
CJ442CG HDWCFG.UP CJ442RAC.CT CJ442RAC.S
```

The ROM Access Module *CJ442RAC.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 *CJ442RAC.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 *CJ442.A*.

Note that you must still include the AMX Library *CJ442.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 MIPS32™ architecture supports instruction caches and data caches. However, the cache sizes and control methods are MIPS32 implementation dependent. To accommodate the differences, AMX MA32 provides a set of cache management services and a mechanism for adapting those services to changing needs.

The MIPS32 architecture defines a System Control Coprocessor (CP0) with registers which define the instruction and/or data cache characteristics. AMX supports caches controlled by CP0. AMX cache management is restricted to the kernel address space (*kseg0*) controlled by field *K0* in the configuration register (CP0 register 16).

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 MA32 Library for each of the supported cache types. These modules include the low level cache control functions needed to manipulate the instruction and/or data caches. These functions are described in Appendix D.2.

The low level cache control functions are NOT dependent on AMX. They can therefore be used to initialize the caches after a power on reset or software reset and to manipulate the caches prior to launching AMX.

AMX also includes a set of high level cache support functions which make use of the low level functions to manipulate caches. These high level functions can be used by applications without an intimate knowledge of the underlying cache architecture.

## Cache Enable/Disable

AMX includes a set of high level AMX cache support functions *cjcfhwXcache* to enable or disable the instruction and/or data caches. In the process of enabling or disabling the caches, the selected caches are also flushed and invalidated.

These functions use the low level cache control functions to manipulate the selected caches. These high level functions are located in your Target Configuration Module allowing the instruction and data cache sizes to be automatically adjusted according to the processor or architecture identified in your User Parameter File.

The AMX functions *cjcfhwXcache* are described in Appendix B. These functions can be called either before or after AMX is launched.

## Cache Flush and Invalidate

AMX includes a set of high level AMX cache support functions *cjcfhwXflush* to flush and invalidate the instruction and/or data caches without enabling or disabling the caches in the process. Furthermore, these functions do not flush the entire cache. They flush and invalidate only the region specified by you.

The AMX *cjcfhwXflush* functions, also located in the AMX Target Configuration Module, use the low level cache control functions.

### Note

The high level AMX cache support functions do not disable interrupts. It is imperative that these functions be called with interrupts disabled in order to avoid cache conflicts while the caches are being manipulated.

## Cache Initialization

When power is first applied to the MIPS32 processor, 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 MIPS32 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 MIPS32 processor 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 MA32 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 MIPS32 caches. Each module supports a specific cache type. Each cache type is given a name *xxx* identifying a particular MIPS32 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 MIPS32 processor. In most cases, the parameters simply accommodate different cache sizes for each particular MIPS32 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:

<i>0x80000000L</i>	Select the instruction cache
<i>0x40000000L</i>	Select the data cache
<i>0x00000001L</i>	0/1 = disable/enable the selected caches

Parameter *icsize* defines the total size, in bytes, of the instruction cache. Parameter *icparam* is used to identify the instruction cache block (cache line) characteristics.

Parameter *dcsize* defines the total size, in bytes, of the data cache. Parameter *dcparam* is used to identify the data cache block (cache line) characteristics.

When used to **flush** the caches, the *command* bit masks are defined as follows:

<i>0x80000000L</i>	Select the instruction cache
<i>0x40000000L</i>	Select the data cache
<i>0x00000002L</i>	Bit is 1 to flush the selected caches

Parameter *icsize* defines the total size, in bytes, of the memory region to be flushed. Parameter *icparam* is the 32-bit memory address of the region of interest. Parameters *dcsize* and *dcparam* are undefined. If *icsize* is 0, the entire instruction and/or data cache will be flushed.

### Note

The low level AMX cache support functions do not disable interrupts. It is imperative that these functions be called with interrupts disabled in order to avoid cache conflicts while the caches are being manipulated.

The low level AMX cache control functions are located in the following assembly language source files.

`CHM32CAS.C` MIPS32 architecture cache (controlled using `CACHE` instruction)

The following table summarizes the supported cache types and identifies which type should be used for the various MIPS32 implementations.

<b>Processor/ Architecture</b>	<b>Function</b>	<b><i>icsize</i></b>	<b><i>icparam</i></b>	<b><i>dcsize</i></b>	<b><i>dcparam</i></b>
MIPS32 (Note 2)	<code>chm32cache</code>	user defined	user defined	user defined	user defined
MIPS32 MIPS 4Kc MIPS 4Km MIPS 4Kp MIPS 5K	<code>chm32cache</code>	(Note 1)	(Note 1)	(Note 1)	(Note 1)

Note 1: For these processors, the cache sizes (*icsize* and *dcsize*) and the cache line sizes (*icparam* and *dcparam*) are derived automatically by AMX. The parameters are derived from information provided in the coprocessor 0 configuration register. The cache configuration parameters are decoded based on the processor or architecture identified in your Target Parameter File. The parameters are derived by procedure `cjcfhwpcache()` in the AMX Target Configuration Module.

Note 2: You can edit your Target Parameter File (see Appendix D.3) to define alternate cache parameters. You must use this technique to accommodate custom ASICs based on the MIPS32 architecture.

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 M32 Cache Services

For most MIPS32 processors, the instruction and data cache are manipulated using the MIPS32 *CACHE* instruction provided for that purpose. The MIPS32 memory management unit (MMU) is implemented as a collection of registers in CP0. These MMU registers define the regions of memory to which instruction and data caching apply. These registers must be initialized by you to match your memory system and to meet the needs of your application. The MMU registers must be initialized before any AMX cache service functions are used.

AMX cache management is restricted to the kernel address space (*kseg0*) controlled by field *K0* in the configuration register (CP0 register 16).

The cache control function *chm32cache* is used to flush, invalidate and enable/disable the instruction and/or data caches as indicated by its *command* parameter. The instruction and data caches can be flushed and invalidated separately. However, when enabling or disabling the cache, the *kseg0* instruction and data caches are both enabled or both disabled.

When flushing caches, the cache control function *chm32cache* operates as follows. If the instruction cache is selected by parameter *command*, it is invalidated. If the data cache is selected, it is flushed and then invalidated.

When disabling the *kseg0* cache subsystem, the cache control function *chm32cache* operates as follows. If either cache is selected by parameter *command*, both caches will be disabled. First, the entire data cache is flushed and then invalidated. The *kseg0* cache (instruction and data) is then disabled. The entire data cache is then flushed and invalidated again. Finally, the entire instruction cache is invalidated.

When enabling the *kseg0* cache subsystem, the cache control function *chm32cache* operates as follows. If either cache is selected by parameter *command*, both caches will be enabled. First, the entire data cache is flushed and then invalidated. Next, the entire instruction cache is invalidated. The *kseg0* cache (instruction and data) is then enabled.

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 or data cache line respectively.

The board initialization function *chbrdinit* in the board support module *MALTA4KC.S* for the MIPS Malta 4Kc Development Board illustrates the proper use of the MIPS32 cache control functions. Function *chbrdinit* is called from the AMX Sample Program *main* function. *Chbrdinit* includes a call to *chm32cache* to initialize the MIPS32 caches.

Note that the call to *chm32cache*, 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 *MALTA4KC.S* to allow the call to *chm32cache*.

This page left blank intentionally.

### D.3 Customizing AMX Cache Services

Unfortunately, although the MIPS32 architecture does define how caches should be implemented and manipulated, some MIPS32 processors may implement a different approach to cache management.

The cache services provided by AMX accommodate the common cache control mechanisms employed by MIPS in its MIPS32 products.

At the time AMX MA32 was first released, all cache management schemes adhered to the MIPS32 architecture specification. Most MIPS32 processors will use this method but may change the cache sizes to meet the needs of particular applications.

To meet these changing requirements, KADAK has provided a cache override facility. To use this feature, edit your Target Parameter File using the AMX Configuration Manager as described in Chapter 4. Make the Target Configuration Module the active selector and go to the Cache property page. Check the box labeled Use custom cache control. In the field labeled Cache function name, enter the name of the low level AMX cache control function *chXXXcache*. Then adjust the cache parameters which will be passed to that function.

For example, to accommodate a 16Kb instruction cache size and an 8Kb data cache size in an MIPS32 processor which incorporates the cache control mechanism found in the MIPS 5K core, adjust the custom cache parameters as follows.

<i>chm32cache</i>	Cache function name	(in AMX Library)
<i>16384</i>	Instruction cache size	(16K bytes)
<i>0x0020</i>	Instruction cache parameter	(32 bytes/line)
<i>8192</i>	Data cache size	(8K bytes)
<i>0x0020</i>	Data cache parameter	(32 bytes/line)

The AMX Configuration Manager inserts a `...CACHE` cache override directive into your Target Parameter File. The `...CACHE` directive is described in Appendix A.2. If you are unable to use the AMX Configuration Manager, you will have to use a text editor to edit your Target Parameter File to include this directive. The `...CACHE` directive from the above example is as follows.

```
...CACHE chm32cache,16384,0x0020,8192,0x0020
```

#### Note

The cache parameters in the `...CACHE` directive must be provided in a form acceptable to the MIPS32 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.

<i>YOURcache</i>	Cache function name	(your cache procedure)
16384	Instruction cache size	(16K bytes)
16	Instruction cache parameter	(user defined)
8192	Data cache size	(8K bytes)
16	Data cache parameter	(user defined)

The resulting `...CACHE` directive will be as follows.

```
...CACHE YOURcache,16384,16,8192,16
```

Your cache control function must be prototyped just like the AMX *chXXXcache* functions.

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

The high level AMX cache service functions *cjcfhwXcache* will automatically be adjusted to call your cache control function *YOURcache* with the four parameters defined in the `...CACHE` directive. The interpretation of these parameters is entirely up to your procedure *YOURcache*. The *command* parameter will adhere to the standard bit mask values supported by all *chXXXcache* functions (see Appendix D.2).

If you use one of the AMX *chbrdinit* board support functions, you will have to edit it to call your new function *YOURcache*.

### Suppressing Low Level Cache Control Services

You can unconditionally suppress the low level AMX cache control functions from your AMX system. To do so, edit your Target Parameter File using the AMX Configuration Manager as described in Chapter 4. Make the Target Configuration Module the active selector and go to the Cache property page. Check the box labeled Suppress all cache support.

If you are unable to use the AMX Configuration Manager, you will have to use a text editor to edit your Target Parameter File to include the following directive in your Target Parameter File.

```
...CACHE NOCACHE
```

## Appendix E. Interrupt Management

### E.1 Interrupt Prioritization and Nesting

The MIPS32 architecture does not prioritize the six hardware interrupt exceptions. It is left to software to establish the order of priority according to the order in which the IP bits in the CP0 cause register are examined.

AMX MA32 establishes an interrupt ordering in which  $IP[7]$  is the highest priority and  $IP[0]$  is the lowest priority. For hardware interrupts, Hw\_Int5 ( $IP[7]$ ) is of highest priority and Hw\_Int0 ( $IP[2]$ ) is of lowest priority. Software interrupts 1 and 0 ( $IP[1]$  and  $IP[0]$ ) are lowest in priority, even if used to simulate device interrupts. This AMX priority ordering is not adjustable.

AMX establishes its priority ordering of interrupts using an interrupt disable mask implemented in software. The AMX priority scheme is described in Chapter 3.3.

Another type of prioritization must occur when multiple devices can generate an interrupt request through one of the MIPS32 hardware interrupt exceptions. The interrupt source must be identified by software with or without the assistance of an interrupt controller. The order in which the interrupt sources are identified determines the order in which the devices are serviced. This order of service establishes the device priority.

If an interrupt controller which supports software or hardware interrupt masking is used, then a further level of prioritization is possible. In this case, service of one device can be preempted in favor of another device of higher priority, even if both devices generate interrupt requests via the same hardware interrupt exception.

AMX MA32 supports all of these types of interrupt prioritization and nesting.

#### Dedicated Interrupt Priorities

In most AMX applications, the six levels of hardware priority established by AMX will be sufficient. Each hardware interrupt is dedicated to a single interrupt source. AMX establishes the order in which interrupts are serviced. The AMX interrupt disable mask can be used to allow interrupt nesting if required. Each interrupt is serviced with all interrupts of equal or lower priority inhibited.

## **Software Prioritization of Multiplexed Interrupts**

When multiple devices are multiplexed through one hardware interrupt exception, the AMX exception handler calls your Interrupt Identification Procedure (IIP) to determine the source of the interrupt. If the IIP must poll the devices to determine the source of the interrupt, the order in which the devices are polled determines the device priority.

If a simple interrupt controller can be used to identify the device requesting service, the order in which the device interrupt request bits in the controller's interrupt identification register are examined determines the device priority.

If the order of priority must be determined by software, the hardware necessary to inhibit device interrupt requests is usually not present. Without some form of interrupt masking, all requests for service at or below a particular priority level cannot be inhibited. Hence, interrupt nesting of the multiplexed devices cannot be supported. Each interrupt request must be serviced to completion before another interrupt via the same hardware interrupt exception can be serviced.

## **Hardware Prioritization of Multiplexed Interrupts**

An external device such as an Intel 8259 Programmable Interrupt Controller (PIC) can be used to arbitrate interrupt requests and assign interrupt priorities to multiple devices. A single 8259 PIC can support as many as eight devices connected to a MIPS32 hardware interrupt request pin.

If more than eight interrupt sources are required, two 8259 PICs can be used, each one connected to a different hardware interrupt request pin. However, it may be more advantageous, and just as effective, to connect one master 8259 PIC to hardware interrupt request pin and attach to it one or more slave 8259 PICs. This configuration achieves full interrupt prioritization, allows nested interrupts and still leaves the remaining five hardware interrupt exceptions free for dedicated use.

When multiple devices are multiplexed through a hardware interrupt exception, the AMX exception handler calls your Interrupt Identification Procedure (IIP) to determine the source of the interrupt. Your IIP must interrogate the controller to determine the highest priority device requesting service. Of course, how this is done will be dictated by the manner in which the 8259 PIC is interfaced to the MIPS32 processor.

## E.2 Nested Interrupts

### Nesting of Hardware Interrupts

AMX MA32 supports nested hardware interrupts. The AMX interrupt disable mask associated with a particular ISP root disables all interrupts of equal or lower priority than the priority of the hardware interrupt being serviced. For a multiplexed hardware interrupt, all of the multiplexed device interrupts share the same AMX priority level.

Each device Interrupt Handler executes with the processor interrupt system enabled ( $IE=1$  in the processor status register). However, interrupts of equal or lower AMX priority are disabled by virtue of the AMX interrupt disable mask. The device Interrupt Handler must clear the source of the interrupt request prior to returning to the ISP root. If an interrupt controller is used to arbitrate a set of multiplexed devices, the device Interrupt Handler must also force the interrupt controller to dismiss the interrupt being serviced.

### Nesting Within a Multiplexed Interrupt

AMX MA32 also supports the nesting of interrupts within a multiplexed hardware interrupt, provided that an interrupt controller or hardware interface is available to arbitrate the requests for service.

If a hardware interrupt exception is configured to support interrupt nesting, the Interrupt Identification Procedure must operate as described in Appendix E.3.

The AMX exception handler will call the IIP with the AMX interrupt disable mask set to disable the multiplexed hardware interrupt being serviced and all lower priority interrupts. If the IIP returns a valid device number, AMX will enable the multiplexed hardware interrupt before calling the device Interrupt Handler.

Each device Interrupt Handler executes with the processor interrupt system enabled ( $IE=1$  in the processor status register). In this case, although interrupts of lower AMX priority are disabled by virtue of the AMX interrupt disable mask, interrupts of equal AMX priority are enabled. The interrupt controller inhibits the interrupt being serviced and all interrupts of lower priority. However, interrupts from higher priority devices are still permitted.

The device Interrupt Handler must clear the source of the interrupt request prior to returning to the ISP root. However, it **must not** manipulate the interrupt controller to dismiss the interrupt being serviced. That is the responsibility of the IIP when nested interrupts are allowed through a multiplexed hardware interrupt.

When the device Interrupt Handler returns to the AMX exception handler, AMX will again inhibit the multiplexed hardware interrupt and call the IIP to force the interrupt controller to dismiss the interrupt being serviced.

### E.3 Interrupt Identification Procedure for Nested Interrupts

An interrupt request from a multiplexed device cannot preempt service of another multiplexed device unless your Interrupt Identification Procedure (IIP) supports nested interrupts.

The Interrupt Identification Procedure for a single multiplexed hardware interrupt exception will be described. The discussion of nesting applies only to interrupt nesting by the devices which share that hardware interrupt. However, since AMX supports a mixture of multiplexed and dedicated hardware interrupt exceptions, each operating at its own AMX priority level, hardware interrupts of higher priority can always preempt service of the multiplexed interrupt being described.

The AMX Configuration Manager inserts a `...ISPMUX` directive into your AMX Target Parameter File to indicate that a particular hardware interrupt exception is to be multiplexed. The directive also specifies if interrupt nesting is to be supported by your IIP. The IIP calling sequence is described in Chapter 3.2. When nesting is supported, the calling sequence is enhanced to allow your IIP to provide both interrupt identification prior to entry to your Interrupt Handler and interrupt dismissal after service is complete.

Your Interrupt Identification Procedure is called by the AMX exception handler whenever a new hardware interrupt is detected. On entry to the IIP, register `v0` is `0` indicating that a new interrupt is about to be serviced. The IIP must record the priority of the interrupt currently under service, if any, and identify the new interrupt source and its priority. The IIP must then inhibit all other interrupts of priority equal to or less than the new interrupt priority. How this is done depends on the nature of the supporting interrupt controller and device interfaces.

When nested interrupts are supported, your Interrupt Identification Procedure is also called by the AMX exception handler whenever your device Interrupt Handler finishes service of an interrupt from that device. On entry to the IIP, register `v0` is `1` (non-zero) indicating that interrupt service is complete. The IIP must restore the priority of the interrupt subsystem to reflect conditions prior to the interrupt which has just been serviced. The IIP must inhibit all other interrupts of priority equal to or less than the restored interrupt priority. Again, how this is done depends on the nature of the supporting interrupt controller and device interfaces.

Most IIPs of this kind will require nested storage to retain the interrupt mask history in order to successfully unravel the nested interrupts. Consequently, for each interrupt, AMX allocates a block of storage, called a history frame, on the AMX Interrupt Stack. Your IIP can use this history frame to save and restore interrupt mask information on entry and exit from each interrupt.

#### Target Parameter File

For your IIP to support nesting, you must first edit your Target Parameter File using the AMX Configuration Manager. Make the Target Configuration Module the active selector and go to the AMX Vectors property page. Select the AMX vector for the multiplexed hardware interrupt of interest and check the box labeled Prioritized/nested by software. In the field labeled IIP storage (bytes), enter the amount of history storage that your IIP requires.

## Interrupt Identification Procedure Calling Convention

When nested interrupts are supported, the following conditions exist upon entry to your Interrupt Identification Procedure.

Interrupts are enabled ( $IE=1$  and  $EPL=EXL=0$ ) but interrupts are masked off according to the interrupt priority established by your definition of the multiplexed hardware interrupt exception.

The processor is executing in kernel mode.

The return address is in register  $ra$ .

The stack pointer in register  $sp$  references the AMX Interrupt Stack.

Register  $v0$  is  $0$  if a new interrupt is to be identified.

Register  $v0$  is  $1$  if interrupt service has been completed.

Registers  $at$ ,  $v0$  and  $v1$  are free for use.

All other registers must be preserved.

Your AMX Target Parameter File defines the size ( $vNEST$  bytes) of the history frame required by your Interrupt Identification Procedure to support prioritized interrupt nesting.

The AMX Interrupt Supervisor allocates a history frame of  $vNEST$  bytes on its stack prior to calling the IIP. The storage frame is therefore unique for each interrupt. Most IIPs will use the frame to store enough interrupt priority information to permit the state of the interrupt subsystem to be restored after service of an interrupt.

For example, the IIP provided with AMX for use with the Intel 8259 compatible interrupt controller on the MIPS 4Kc Malta Development Board requires only 4 bytes of storage. When called upon to identify the source of an interrupt, this IIP saves an end-of-interrupt (EOI) identifier in the dedicated history frame. The device requesting service is then identified. Interrupts from all lower priority devices are automatically inhibited by the interrupt controller. Once the interrupt has been serviced, the IIP issues the EOI to the interrupt controller using the information retrieved from the dedicated history frame.

## Interrupt Identification Procedure Return Values

The Interrupt Identification Procedure must return the interrupt identification number for the particular device which generated the exception. The interrupt identification number is an index ( $0$  to  $n-1$ ) into a block of  $n$  vectors in the AMX Vector Table allocated to the devices which are multiplexed through the hardware interrupt exception.

The interrupt identification number is returned in register  $v0$  as follows:

$v0 = i$	Interrupt number ( $0$ to $n-1$ )
$v0 = -1$	Ignore the interrupt
$v0 = -2$	Generate a fatal exception (unidentified interrupt request)

If the IIP returns  $v0 = -1$  or  $-2$ , the interrupt mask (or its equivalent) used for prioritizing device interrupts should not be altered. In either case, the AMX exception handler will ignore the device and will not signal an end of interrupt with a second call to the IIP. As a result, any information saved by the IIP in its private frame on the AMX Interrupt Stack will be lost.

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