AMX®

AMX™ 386/ET Target Guide

First Printing: **October 15, 2001**
Last Printing: **November 1, 2007**

TECHNICAL SUPPORT

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

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

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

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

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

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

DISCLAIMER

TRADEMARKS

AMX 386/ET TARGET GUIDE
Table of Contents

AMX 386/ET TARGET GUIDE
Table of Contents (Cont'd)

Appendices

AMX 386/ET TARGET GUIDE
Table of Figures

1. Getting Started with AMX 386/ET

1.1 Introduction

The AMX™ Multitasking Executive is described in the AMX User's Guide. This target guide describes AMX 386/ET which operates on the Intel386™, Intel486™, Pentium™ and all architecturally compatible processors.

Throughout this manual, the term i386 refers specifically to the Intel386, Intel486 and Pentium families of processors and all processors which are exact replicas. When distinctions are not important, the term i386 is used to reference any processor which has the general characteristics of these families. 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 386/ET real-time system. It is assumed that you have read the AMX User's Guide and are familiar with the architecture of the i386 processor.

Installation

AMX 386/ET is delivered ready for 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 386/ET has been tested by KADAK, a separate chapter in the **AMX 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 i386 assembly language.

Figures 1.2-1, 2 and 3 summarize the AMX modules provided with AMX 386/ET. The AMX product manifest (file *MANIFEST.TXT*) is a text file which indicates the current AMX revision level and lists the AMX modules which are provided with the product.

File Name	Module
CJ722 .H	Generic include file
CJ722APP.H	Custom application definitions
CJ722CC .H	C dependent definitions
CJ722EC .H	AMX error code definitions
CJ722IF .H	C and target interface prototypes
CJ722KC .H	Private AMX constants
CJ722KF .H	AMX service procedure prototypes
CJ722KP .H	Private AMX prototypes
CJ722KS .H	Private AMX structure definitions
CJ722KT .H	Target processor definitions
CJ722KV .H	AMX version specification
CJ722SD .H	AMX application structure definitions
CJ722TF .H	Target dependent prototypes
CJZZZ .H	Copy of generic include file *CJ722.H* used for portability
CHxxxxx .H	Definitions for common timer (PIT) and serial I/O (UART) chips

Figure 1.2-1 AMX Include Files

File Name	Module
CJ722K .DEF	Private AMX assembly language definitions
CJ722KQ .ASM	Private AMX math procedures
CJ722KR .ASM	AMX Interrupt Supervisor
CJ722KS .ASM	AMX Task Scheduler
CJ722MXA.ASM	Message Exchange Manager constants
CJ722TDC.ASM	Time/Date Manager constants
CJ722UA .ASM	Target processor and C support (part 1)
CJ722UB .ASM	Target processor and C support (part 2)
CJ722BSC.ASM	i386 Bootstrap example
CJ722BSU.ASM	i386 Bootstrap utilities

Figure 1.2-2 AMX Assembler Source Files

File Name	Module
CJ722KA .C	Kernel task services
CJ722KB .C	General task services
CJ722KBR.C	
CJ722KC .C	Timer Manager
CJ722KCR.C	
CJ722KD .C	Task management services
CJ722KDR.C	
CJ722KE .C	Task termination services
CJ722KF .C	Suspend/resume task
CJ722KG .C	Time slice services
CJ722KH .C	Task status
CJ722KI .C	Enter and Exit AMX
CJ722KJ .C	General object access
CJ722KK .C	AMX Vector Table access
CJ722KL .C	Private AMX list manipulation
CJ722KM .C	AMX task scheduler hook services
CJ722KX .C	AMX Kernel Task
CJ722CL .C	Circular List Manager
CJ722LM .C	Linked List Manager
CJ722BM .C	Buffer Manager
CJ722BMR.C	
CJ722EM .C	Event Manager
CJ722EMR.C	
CJ722RM .C	Semaphore Manager (resources)
CJ722SM .C	Semaphore Manager
CJ722SMR.C	
CJ722MB .C	Mailbox Manager
CJ722MBR.C	
CJ722MF .C	Flush mailbox and message exchange
CJ722MM .C	Memory Manager
CJ722MMR.C	
CJ722MX .C	Message Exchange Manager
CJ722MXR.C	
CJ722TDA.C	Time/Date Manager
CJ722TDB.C	Time/Date formatter
CJ722UF .C	Launch and leave AMX
CJ722XTA.C	Message exchange task services
CJ722XTB.C	Message exchange task termination
CHxxxxxT.C	Clock drivers for common timer (PIT) chips
CHxxxxxS.C	Sample drivers for common serial I/O (UART) chips
AT386BRD.C	i386 PC/AT board support module

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` or `0ABCDH`.

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

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

AMX symbol names and reserved words are identified as follows:

`cjkkpppp`	AMX C procedure name `pppp` for service of class `kk`
`cjxtttt`	AMX structure name of type `tttt`
`xttttyyy`	Member `yyy` of an AMX structure of type `tttt`
`CJ_ID`	AMX object identifier (handle)
`CJ_ERRST`	Completion status returned by AMX service procedures
`CJ_CCPP`	Procedures use C parameter passing conventions
`CJ_ssssss`	Reserved symbols defined in AMX header files
`CJ_ERxxxx`	AMX Error Code `xxxx`
`CJ_WRxxxx`	AMX Warning Code `xxxx`
`CJ_FExxxx`	AMX Fatal Exit Code `xxxx`
`CJ722xxx.xxx`	AMX 386/ET filenames
`CJZZZ.H`	Generic AMX include file

The generic include file `CJZZZ.H` is a copy of file `CJ722.H` which includes the subset of the AMX 386/ET 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 i386 processor.

Processor registers are referenced using the software names specified by Intel.

```
EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP
DS, ES, FS, GS, CS, SS
EFLAGS = flags register, CC = status flags (condition code)
```

1.4 AMX 386/ET Target Specifications

AMX 386/ET was initially developed and tested using the Intel386, Intel486 and Pentium processors on a variety of i386 PC/AT platforms. However, the AMX 386/ET design criteria fully encompass the Intel i386 processor family requirements.

AMX uses a set of design constants which vary according to the constraints imposed by each target processor. When operating on the i386 processor, these design constants assume the values listed in Figure 1.4-1.

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

Figure 1.4-1 AMX Design Constants

1.5 Launch Requirements

The i386 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.

It is assumed that you will have a boot ROM present which configures the i386 for your specific hardware configuration and begins program execution at the entry to your C startup code. AMX includes sample bootstrap code to initialize the i386 processor (see Appendix D).

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 operating in protected mode at privilege level 0. The default state when the processor is reset is real mode. Your target hardware ROM monitor must initialize the i386 global, local and interrupt descriptor tables and switch the processor from real mode to protected mode before launching AMX. The AMX 386/ET bootstrap code described in Appendix D illustrates one method of providing the switch to protected mode.

Interrupt State

Interrupts can be enabled or disabled on entry to AMX. Set the interrupt enable flag (`IF`) in the flags register to 0/1 to disable/enable external interrupts. AMX will disable interrupts during its startup initialization. AMX will enable interrupts prior to calling your application Restart Procedures.

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

Direction Flag

AMX follows the convention adopted by most C compilers that the direction flag (*DF*) in the flags register will always be set to 0 for low to high addressing. AMX always clears the direction flag before calling any application procedure. The direction flag MUST be clear upon entry to any AMX procedure which is coded in C.

i386 Stack Use

The i386 begins execution in real mode with NO stack. Your ROM monitor must establish a valid stack. Your bootstrap code or C startup code may switch to an alternate stack. Once AMX is launched, it abandons the startup stack. AMX only uses the stacks allocated by you in your AMX System Configuration Module.

Instruction and Data Caching

The Intel486 includes an 8192-byte cache used for instruction and data cache.
The Pentium includes an 8192-byte instruction cache and an 8192-byte data cache.

If your AMX Target Parameter File (see Chapter 4) targets one of these processors, AMX will automatically flush and enable both caches when AMX is launched. Alternatively, you can configure AMX to ignore the caches during the launch. AMX provides procedures which you can use to enable or disable the caches.

For example, if you disable both 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.

You must be aware that, on processors which utilize an i386 Memory Management Unit (MMU), successful cache operation will depend on proper setup of the MMU. AMX does not manipulate the MMU. If you configure AMX to enable caching during the launch, then you must ensure that the MMU is properly initialized to meet your hardware memory addressing specifications prior to launching AMX. The AMX Sample Program purposely leaves the caches unaltered to avoid possible cache related problems during your initial use of AMX in your hardware environment.

Memory Management Unit (MMU)

The Intel386, Intel486 and Pentium include a Memory Management Unit (MMU) to support a demand-paged virtual memory environment. AMX does not support the i386 memory management unit.

Your AMX application code and data must reside within the memory address ranges allowed by the particular i386 processor which you are using. The i386 MMU, if present, must be setup prior to launching AMX. In most cases, your boot ROM or C startup code will configure the i386 MMU for your specific hardware configuration prior to entry to your `main()` program.

Warning!

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

Big or Little Endian

AMX 386/ET adheres to the little endian model in which the least significant byte of a word (long) is stored in the lowest byte address.

Be aware that AMX for other processors may be big or little endian. If you intend to port your AMX application to other processors, then avoid using coding techniques which are endian dependent.

2. Program Coding Specifications

2.1 Task Trap Handler

AMX 386/ET supports task traps for the i386 zero divide, bounds check and overflow
faults. A zero divide fault occurs if any i386 instruction attempts an integer division by
zero. A bounds check fault occurs if the i386 *BOUND* instruction detects an array bound
violation. An overflow fault occurs if the overflow flag (*OF*) is set in the flags register
(*EFLAGS*) at the time an i386 *INTO* instruction is executed.

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

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

void CJ_CCPP traphandler(
struct cjxregs *regp,                        /* A(Register Structure)    */
void         *faultfp)                       /* A(Fault frame)           */
{
   :
   Process the error
   :
   }
```

The zero divide, bounds check and overflow exceptions are serviced by AMX. 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 *CJ722KT.H*.

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

A pointer to the i386 fault frame is provided as parameter *faultfp*. This pointer is the
i386 stack pointer (*ESP*) after the fault has occurred. Fault frame members can be
referenced as described in Chapter 3.1.

The register values in structure *regs* can be examined and, in rare circumstances,
modified. If necessary, the fault frame at **faultfp* 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 determined by the fault frame at
*faultfp with registers set according to the values in the structure referenced by *regp*.
Note that the *EFLAGS* register will be restored according to the value returned in the fault
frame referenced by *faultfp*.

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.
> Registers `DS = SS = DGROUP` selector.
> The Task Control Block address is in register `DS:ESI`.
> The stack pointer in register `SS:ESP` references the task's stack.
> The return address is on the stack at `[ESP]`.
> Registers `EAX`, `EBX`, `ECX` and `EDX` are free for use.
> Condition code flags in the flags register (`EFLAGS`) can be altered.
> All other registers must be preserved.

Your procedures receive a pointer to the Task Control Block (TCB) of the task which is being started, ended, suspended or resumed. If you include AMX header file `CJ722K.DEF` in your assembly language module, you can reference the private region within the TCB reserved for your use as `[ESI].XTCBUSER`.

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

3. The Processor Interrupt System

3.1 Operation

The i386 classifies all internal and external sources of interruption as interrupts or exceptions. The processor automatically determines the cause of the interrupt or exception and then branches indirectly through entries in the processor Interrupt Descriptor Table to an appropriate interrupt or exception specific procedure.

The particular procedures which service internal or external device interrupt requests are called **Interrupt Service Procedures**. All other procedures are referred to as **exception service procedures**.

Upon entry to any Interrupt Service Procedure or exception service procedure the processor state is determined by the particular exception.

Device Interrupt Service

A subset of the exception vectors are reserved for the control of devices external to, or embedded in, the processor. These vectors include:

Vector	2	Non-maskable interrupt vector
Vectors	32	User assignable interrupts
to	255	

The external interrupt facility is enabled by setting the interrupt flag (IF) in the processor flags register ($EFLAGS$) to 1.

The external interrupt facility is disabled by setting the interrupt flag (IF) in the processor flags register ($EFLAGS$) to 0. Note that the non-maskable interrupt cannot be inhibited.

When an interrupt occurs, the processor pushes the current content of the processor $EFLAGS$ register onto the current stack. The return address (current Instruction Pointer) is then pushed onto the current stack. The processor interrupt flag (IF) in the processor $EFLAGS$ register is set to 0 thereby disabling all external interrupts.

The interrupting device then identifies the interrupt source by presenting the processor with its vector number. Any vector number in the range 0 to 255 is possible, but vectors 32 to 255 are reserved for this purpose. Programmable devices which have not been programmed with their particular vector number can be expected to produce bizarre effects.

Default Exception Service Procedures

AMX provides default service procedures for most exceptions. The zero divide, bounds check and overflow exceptions are serviced by AMX using its Task Trap Handler mechanism. All other exceptions handled by AMX are treated as fatal. AMX calls a Fatal Exception Procedure `cjksfatalexh` in module `CJ722UF.C` identifying the exception and the machine state at the time of the exception. If the Fatal Exception Procedure returns, AMX calls the Fatal Exit Procedure `cjksfatal` in the same module with one of the following fatal exit codes:

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

The **Fatal Exception Procedure** is written in C as follows. Prior to 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        vnum,                                /* Vector number          */
void       *faultfp)                            /* A(Fault frame)         */
{
   :
   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 `CJ722KT.H`.

Note that the `EFLAGS` register copy in the register array reflects the state of the status register after the exception occurred.

A pointer to the i386 fault frame is provided as parameter `faultfp`. This pointer is the i386 stack pointer (`ESP`) after the fault has occurred. Fault frame members can be referenced as follows:

> `*((CJ_T32U *)faultfp)`   is the old `EIP` (or the frame's error code)

Warning!

If the stack selector `SS` in effect at the time of the exception is NOT the flat model data selector used by C for access to data, the pointers `regp` and `faultp` will require a `FAR` pointer `SS` override to reference the information stored on the stack.

3.2 AMX Vector Table

The i386 processor provides an Interrupt Descriptor Table, often referred to as the AMX Vector Table, through which device interrupts are vectored and processor faults are trapped. The position of entries in the table and the vector numbers used to reference them are dictated by Intel.

AMX provides a set of `cjksixxxx` service procedures to allow you to dynamically access or modify entries in the Interrupt Descriptor Table. The Intel identifiers (called vector numbers by AMX) must be used in all calls to these procedures to identify entries in the table.

Device Interrupts

AMX uses the Interrupt Descriptor Table to maintain pointers to Interrupt Service Procedures for all of the device interrupts to which the processor will respond. AMX does not provide a default Interrupt Service Procedure for every device interrupt. However, AMX does provide a default exception service procedure for the non-maskable interrupt (vector number 2).

Processor Exceptions

AMX maintains entries in the Interrupt Descriptor Table for all of the processor exceptions for which AMX assumes responsibility. These entries in the Interrupt Descriptor Table are identified by Intel's exception identifiers (vector numbers) which are defined in AMX header file `CJ722KT.H`. Figure 3.2-1 summarizes the exception vector mnemonics.

A 32-bit mask in your Target Parameter File is used to specify which of the possible exceptions you wish AMX to service. The mask bits are defined in Figure 3.2-1. The AMX Configuration Builder (see Chapter 4) puts a directive in your Target Parameter File to specify the mask required to meet your configuration requirements.

If an enable mask bit is not defined in Figure 3.2-1 for a particular exception, then AMX will not provide a default exception service procedure for that exception.

AMX does not provide default exception service procedures for any of the entries which Intel has declared as undefined but reserved.

Vector Name	Vector Number	Enable Mask	Exception
CJ_PRVNZD	0	00000001H	Divide error
CJ_PRVNDB	1	00000002H	Debug exception
CJ_PRVNNM	2	00000004H	Non-maskable interrupt
CJ_PRVNBP	3	00000008H	Breakpoint (INT 3 instruction)
CJ_PRVNOV	4	00000010H	INTO instruction trap
CJ_PRVNBD	5	00000020H	BOUND instruction trap
CJ_PRVNOP	6	00000040H	Invalid opcode
CJ_PRVNNC	7	00000080H	Coprocessor not available
CJ_PRVNDF	8	00000100H	Double fault
CJ_PRVNCS	9	00000200H	Coprocessor segment overrun
CJ_PRVNTS	10	00000400H	Invalid TSS
CJ_PRVNNS	11	00000800H	Segment not present
CJ_PRVNSF	12	00001000H	Stack exception
CJ_PRVNGP	13	00002000H	General protection
CJ_PRVNPF	14	00004000H	Page fault
	15		reserved
CJ_PRVNCE	16	00010000H	Coprocessor error
CJ_PRVNAL	17	00020000H	Unaligned memory
	18 to 31		reserved
	32 to 255		User defined interrupts

Figure 3.2-1 AMX Vector Table and Vector Numbers

3.3 AMX Interrupt Priority and NMI

The i386 family of processors does not offer inherent interrupt priority ordering. However, many i386 hardware implementations include one or more Intel 8259 Interrupt Controllers which prioritize the interrupt requests from multiple external devices. The AMX Interrupt Supervisor supports this feature and allows the nesting of interrupts for fast response to high priority events.

The i386 interrupt mask (*IF*) in the flags register (*EFLAGS*) establishes the current interrupt state. Tasks run at the lowest interrupt priority level with all interrupt sources enabled (*IF = 1*). Some interrupts may be specifically disabled by an external interrupt controller.

If no external interrupt controller is present, Interrupt Service Procedures run with interrupts disabled (*IF = 0*). The ISP must NOT enable interrupts.

When an external interrupt controller is present, Interrupt Service Procedures run at the interrupt priority level dictated by the manner in which the interrupt source is connected to the interrupt controller. When the interrupt occurs, all interrupts are disabled and remain so until the ISP explicitly enables interrupts by setting *IF = 1*. Lower priority interrupt requests are masked by the interrupt controller. An ISP must NOT instruct the interrupt controller to drop the interrupt priority level to any level below that of the interrupt which it is servicing. When the ISP completes its service, it must disable interrupts (*IF = 0*) and issue an end of interrupt command to the interrupt controller. Only then can the ISP return to the point of interruption.

Non-Maskable Interrupt

The Intel i386 processor provides a non-maskable interrupt (NMI). This interrupt cannot be inhibited by software. The processor will respond to any request on the NMI pin by generating a non-maskable interrupt. When the non-maskable interrupt occurs, the processor automatically saves the processor flags register and the return address on the current stack. The processor then vectors to a memory address determined by the NMI interrupt vector (vector number 2) in the Interrupt Descriptor Table.

You have complete control over the non-maskable interrupt ISP. Usually, the NMI interrupt is used to signal a catastrophic event such as a pending loss of power. The NMI ISP must not use any AMX services. The ISP must process the interrupt in an application-dependent fashion, restore all registers and return to the point of interruption if feasible. This ISP must assure that the interrupt facility is restored according to its state at the time the non-maskable interrupt occurred.

Warning!

Because the occurrence of an NMI interrupt cannot be controlled, the NMI interrupt can occur at any instant, including within critical sections of AMX.

Consequently, the NMI ISP cannot use AMX service procedures for task communication.

3.4 Conforming ISPs

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

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

The ISP root is the actual Interrupt Service Procedure which is executed by the processor when the interrupt occurs. The ISP root calls the AMX Interrupt Supervisor to indicate that interrupt service has begun.

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

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

> Interrupts are disabled.
> The stack pointer in register *SS:ESP* references the AMX Interrupt Stack.
> The data segment register *DS* is set to *DGROUP*.
> All other registers are in the state required by C.

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

> Your Interrupt Handler parameter is in register *EAX*.
> The stack pointer in register *SS:ESP* references the AMX Interrupt Stack.
> The data segment register *DS* is set to *DGROUP*.
> The return address is on the stack at *[ESP]*.
> Registers *EAX*, *EBX*, *ECX* and *EDX* are free for use.
> Condition code flags in the flags register (*EFLAGS*) can be altered.
> All other registers must be preserved.

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

```
/* The ISP root definition in the Target Parameter File is as follows:*/
/*          ...ISPC deviceisp,deviceih,226,0,0                        */
/* The ISP root is given the public name deviceisp                    */
/* The Interrupt Handler is named deviceih                            */
/* The device interrupts on vector number 226                         */


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


/* Assume dcbinfo is some application device control block structure. */
/* Assume deviceXdcb is a structure variable defined as               */
/* "struct dcbinfo deviceXdcb;".                                      */
/*                                                                    */
/* The ISP root definition in the Target Parameter File is as follows:*/
/*          ...ISPC dcb_isp,dcb_ih,230,deviceXdcb,1                   */
/* The ISP root is given the public name dcb_isp                      */
/* The Interrupt Handler is named dcb_ih                              */
/* The device interrupts on vector number 230                         */
/* deviceXdcb is the name of the public structure variable which      */
/* contains information about the specific device.                    */


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

3.5 Nonconforming ISPs

The i386 family of processors provides an interrupt mechanism which permits the use of nonconforming ISPs within an AMX system. Since nonconforming ISPs bypass the AMX Interrupt Supervisor, they cannot make use of any AMX services.

Upon entry to a nonconforming ISP the processor state matches its state at the time of the interrupt. All interrupts are disabled. No registers are free for use. All registers must be preserved.

The nonconforming ISP executes on the stack in effect at the time of the interrupt. Hence, the nonconforming ISP may execute on any task stack including the AMX Kernel Task's stack. A nonconforming ISP will execute on the AMX Interrupt Stack if the nonconforming ISP interrupts a conforming ISP.

The nonconforming ISP must service the device to remove the interrupt request and dismiss the interrupt with an *IRETD* instruction.

If no external interrupt controller is present, the nonconforming ISP runs with interrupts disabled. The ISP must NOT enable interrupts.

When an external interrupt controller is present, the nonconforming ISP runs at the interrupt priority level dictated by the manner in which the interrupt source is connected to the interrupt controller. When the interrupt occurs, all interrupts are disabled and remain so until the ISP explicitly enables interrupts by setting *IF = 1* in the flags register. Lower priority interrupt requests are masked by the interrupt controller.

A nonconforming ISP must NOT instruct the interrupt controller to drop the interrupt priority level to any level below that of the interrupt which it is servicing. Higher priority interrupts are only allowed if the corresponding ISPs are also nonconforming ISPs.

A nonconforming ISP must NOT allow an interrupt from ANY higher priority conforming ISP. Remember that, in this context, the ISP for the device which generates the AMX clock interrupt is considered to be a conforming ISP.

When the ISP completes its service, it must disable interrupts (*IF = 0*) and issue an end of interrupt command to the interrupt controller. Only then can the nonconforming ISP return to the point of interruption with an *IRETD* instruction.

3.6 Processor Vector Initialization

Whenever an internal or external device interrupt occurs, the i386 processor unconditionally vectors to a unique memory address determined by an entry in the processor Interrupt Descriptor Table. The code located at that address is called an Interrupt Service Procedure.

Whenever an exception occurs, the i386 processor also unconditionally vectors to a unique memory address determined by an entry in the processor Interrupt Descriptor Table. The code located at that address is called an exception handler.

Your Target Parameter File defines whether the Interrupt Descriptor Table is in ROM or RAM. The Target Parameter File further qualifies whether or not AMX is allowed to modify the table if it is in RAM.

If the table is declared to be alterable, AMX will allow you to dynamically install pointers to ISPs and exception handlers into the Interrupt Descriptor Table.

If the Interrupt Descriptor Table is in RAM and the table is declared to be alterable, AMX will install pointers to the AMX Exception Supervisor into selected exception vectors in the Interrupt Descriptor Table.

If the Interrupt Descriptor Table is unalterable (in ROM or simply constant by design), then it is your responsibility to initialize the descriptor table to meet your requirements. The address of a unique AMX exception handler must be installed in each entry in the Interrupt Descriptor Table for which AMX is to be responsible.

Each AMX exception handler is located at an offset from entry point `cj_kdevt` in your Target Configuration Module. Each offset is a multiple of 8 bytes. The AMX exception mask identifies the specific exceptions which AMX must handle. An exception is supported if its mask bit (see Figure 3.2-1) is enabled in the AMX exception mask. The AMX exception handler for the exception identified by mask bit `j` is located at byte address `cj_kdevt+(i*8)` where `i` is **one less** than the sum of the enabled bits in the AMX exception mask, counted from bit `0` to bit `j` inclusive.

For example, if vectors 0 (divide error) and 5 (`BOUND` trap) inclusive are the only vectors to be serviced by AMX, the AMX exception mask will have value `0x00000021`. The AMX `BOUND` trap exception handler will be found at entry point `cj_kdevt+(1*8)` (enable mask is `0x0020`, `j` is `5`, the bit sum is `2` and `i` is therefore `1`).

You must also initialize entries in the Interrupt Descriptor Table for each interrupt which your application can generate. For each interrupting device, you must install the address of the device's Interrupt Service Procedure (ISP) into the device's entry in the descriptor table. For each conforming ISP or clock ISP, the address is the pointer to the ISP root named in your AMX Target Configuration Module. For prebuilt AMX clock drivers, you can determine the ISP root name by examining the call to `chclkins()` in procedure `chclockinit()` in the clock driver source module.

The entries in the processor Interrupt Descriptor Table are 8-byte packed descriptors, not simple pointers. The entries encoded in your ROMed Interrupt Descriptor Table for the AMX exception handlers must be defined as trap gates. Those for ISPs must be defined as interrupt gates. All must be at privilege level 0. The ease with which such a ROMed table can be created will depend upon the link and locate tools which you use.

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, `CJ722CM.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.ASM`, 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.
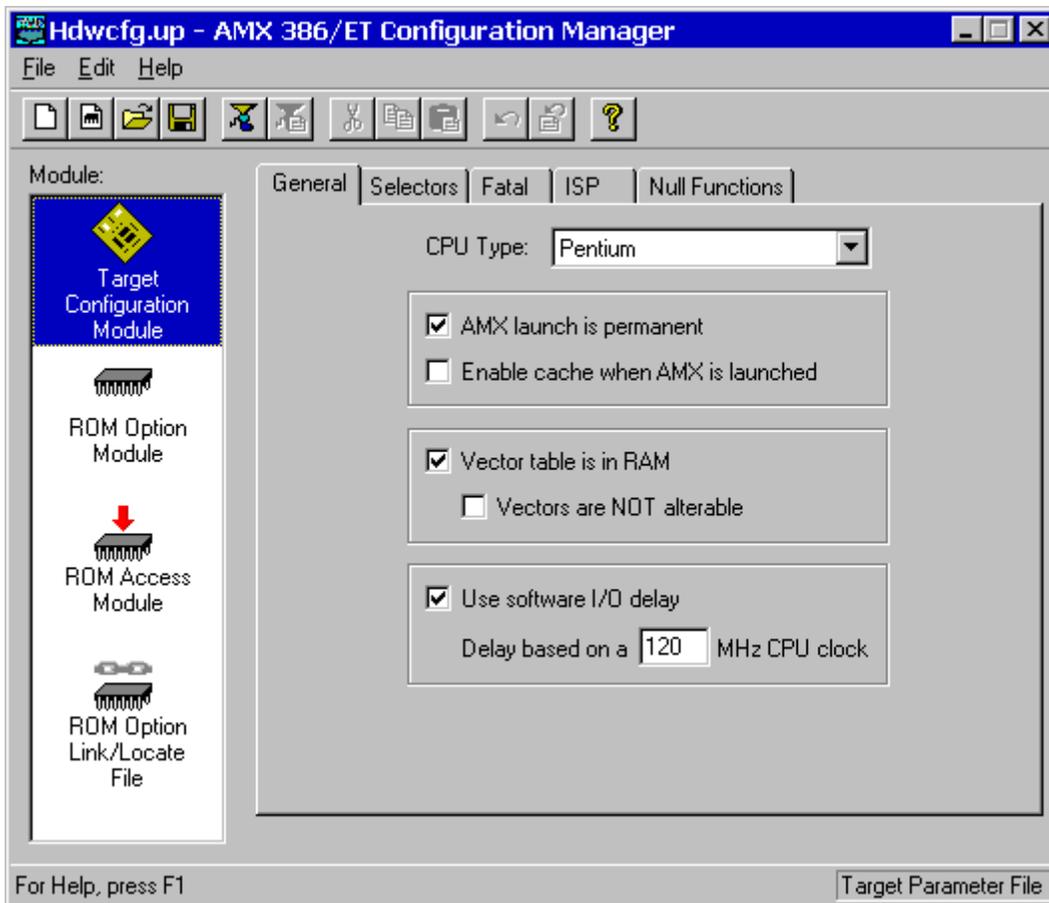
Figure 4.1-1 Configuration Manager Screen Layout

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 supported list of processors includes but is not limited to:

`80386`	{any of Intel386 DX, SX, CX or EX}
`i486`	{any of Intel486 DX2, DX or SX}
`Pentium`	{Pentium, Pentium II, Pentium Pro}

AMX Launch

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

Enable Cache at Launch

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

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

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

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

Vectors in RAM

In most cases, the processor Interrupt Descriptor Table will be located in alterable RAM. Therefore check this box.

If your processor Interrupt Descriptor Table is in ROM, leave this box unchecked. In this case, you must initialize the ROM vector table for AMX use as directed in Chapter 3.6.

Vectors Not Alterable

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

Software I/O Delay

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

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

For example, if your processor executes at 120 MHz with no wait states for instruction fetches and one clock cycle per instruction, enter a CPU clock frequency of 120 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.

Selector Definitions

The Target Configuration Module defines the selectors to be used by your application to access your Interrupt Descriptor Table, *DGROUP* data memory and physical (absolute) memory. These selector definitions are specified by you in the Selector Definition window. The layout of the window is shown below.

The selector values must be derived from your link and locate tool. Most tools give you the ability to identify or allocate specific selectors.
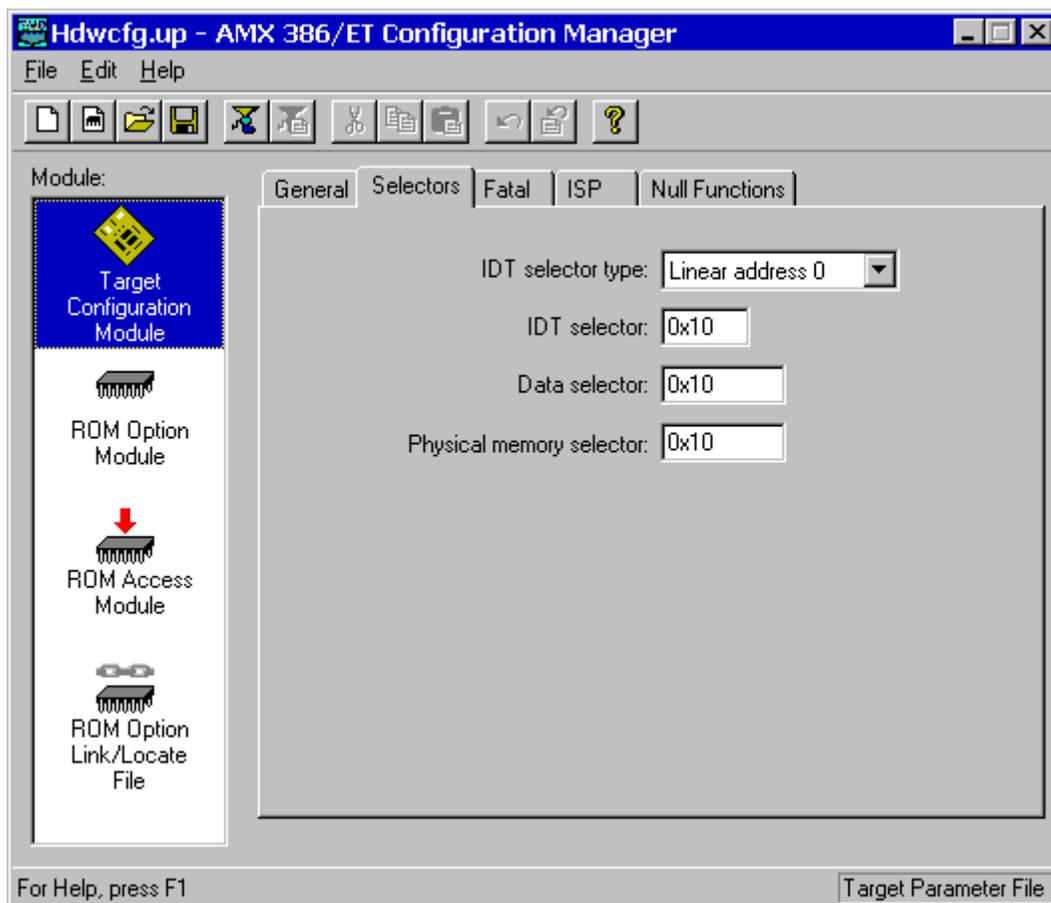
IDT Selector Type

This parameter is used to specify the manner in which the address of the Interrupt Descriptor Table is determined. Two choices are available from the pull down list. The value specified for the IDT selector is dependent upon the IDT selector type which you choose.

IDT Selector

This parameter specifies a particular selector in the Global (or Local) Descriptor Table which can be used to access the Interrupt Descriptor Table. The value must be an integer in the range 0 to $0x1FFC$. The value must be divisible by four.

If you do not know the IDT selector value, set this parameter to 0. Since AMX cannot access the Interrupt Descriptor Table using selector 0, the AMX procedures *cjksidtxxx* will generate error *CJ_ERNOACCESS* indicating that the IDT is not accessible.

If the selector type is the **Actual IDT selector**, this parameter must specify a selector which can be used to directly access the Interrupt Descriptor Table. The Interrupt Descriptor Table is assumed to be addressable at offset 0 using this selector. For example, if the selector value is *0x40*, then the Interrupt Descriptor Table can be accessed by AMX at address *40H:00000000H*.

When the selector type is **Linear address 0**, this parameter must specify a selector which can be used to access memory beginning at linear address 0. The *LIMIT* value in the descriptor must be such that the entire Interrupt Descriptor Table lies within the segment described by the selector. AMX reads the *IDTR* register to extract the linear base address of the Interrupt Descriptor Table, say *IDTADR*. The Interrupt Descriptor Table is assumed to be addressable at offset *IDTADR* using this selector. For example, if the selector value is *0x10* and the extracted linear address is *0x0008C000*, then the Interrupt Descriptor Table can be accessed by AMX at *10H:0008C000H*.

Data Selector

This parameter specifies the run-time selector value corresponding to a descriptor in the Global (or Local) Descriptor Table which is used to access the read/write data segment commonly described as a member of group *DGROUP*. Your AMX application is assumed to execute with this selector value in segment register *DS*. The value must be an integer in the range *0* to *0x1FFC*. The value must be divisible by four.

Physical Memory Selector

This parameter specifies the run-time selector which can be used to access all of physical memory. Memory at physical (absolute) address 0 is assumed to be addressable at offset 0 using this selector. The value must be an integer in the range *0* to *0x1FFC*. The value must be divisible by four. For most toolsets which use the flat memory model, the physical memory selector is the same as the data selector.

Fatal Exceptions

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

This example leaves the debug, non-maskable interrupt, breakpoint, invalid opcode and unaligned memory exceptions free for use by a debugger.

Note that the divide error, *BOUND* and *INTO* exceptions are also serviced by AMX in order to allow the use of Task Trap Handlers by your application tasks. If any of these exceptions occur outside a task or in a task with no Task Trap Handler, AMX will treat the exception as fatal.

In this example, all other exceptions are serviced by AMX and treated as fatal.
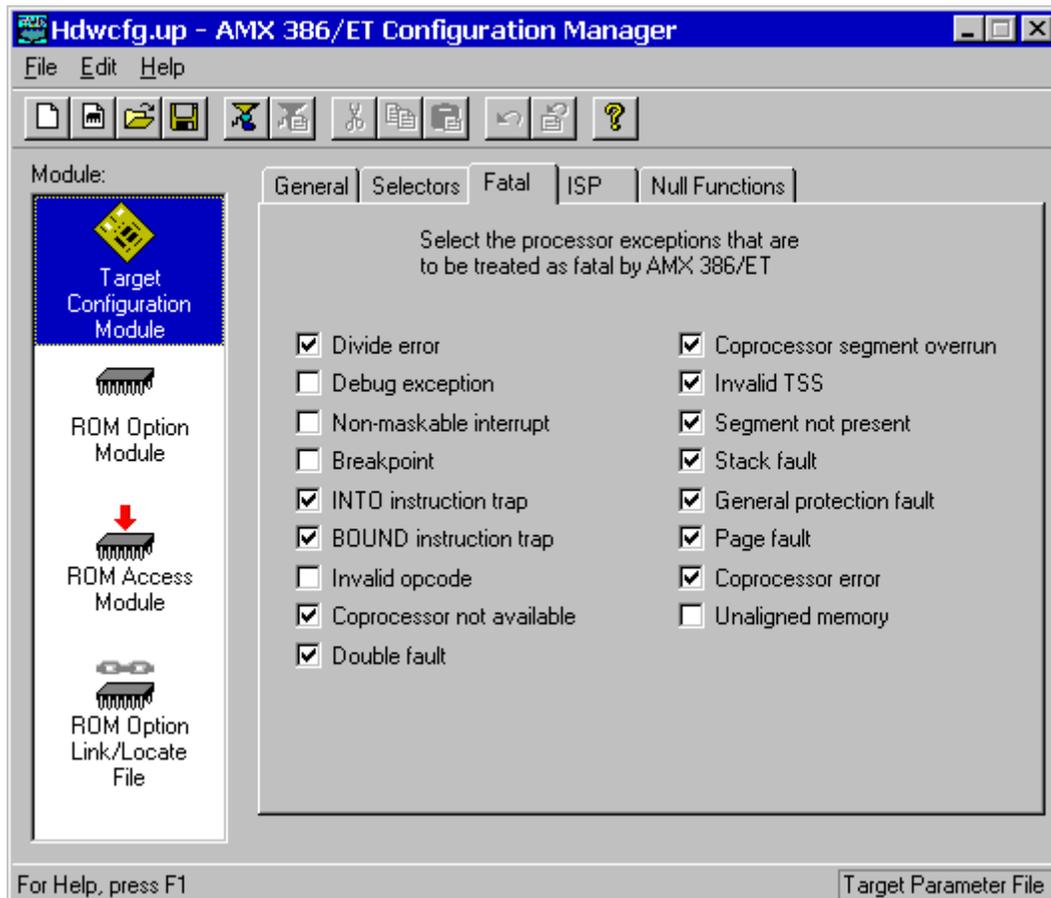
4.3 Interrupt Service Procedure (ISP) Definitions

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

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

To edit an existing ISP definition, double 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.
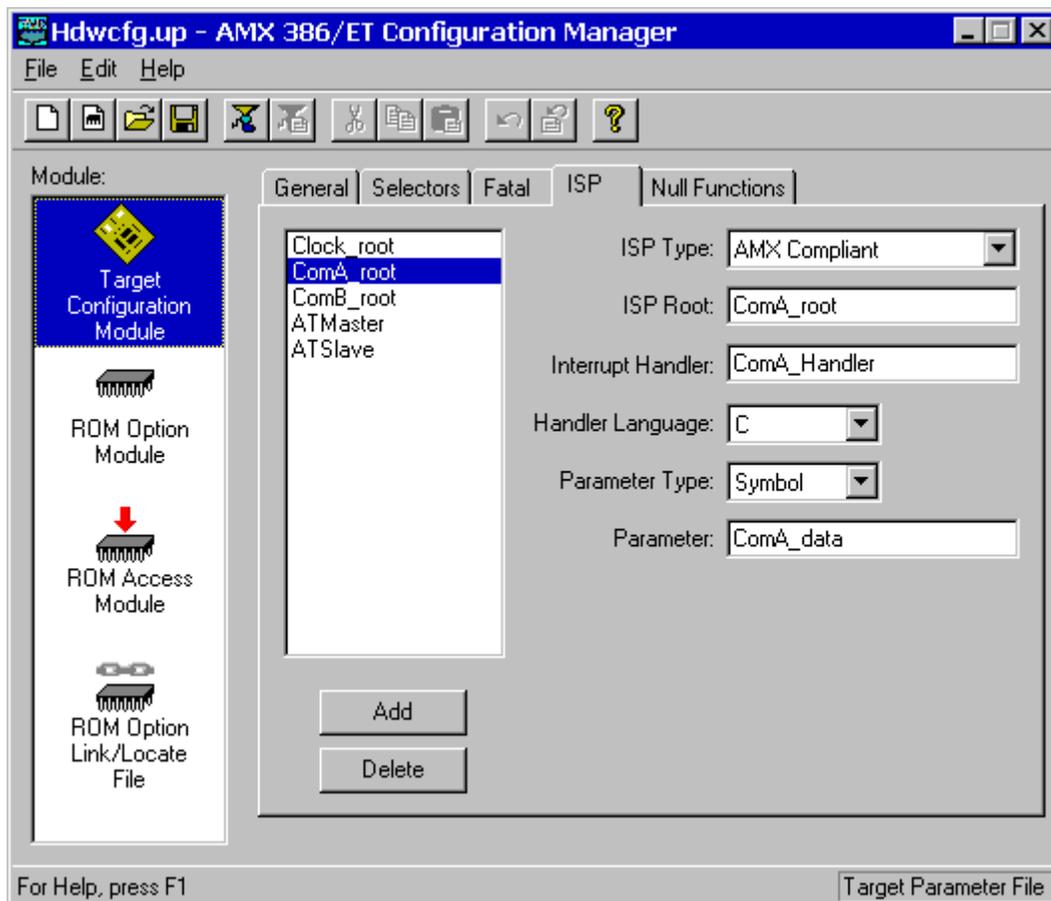
ISP Type

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

If your hardware configuration includes one or more 8259 Interrupt Controllers, then you must be prepared to support the IRQ7 spurious interrupt signal which these controllers can generate. If no physical device is attached to IRQ7 on a master 8259, you can choose 8259 Spurious (master) to allow the predefined AMX handler to service the interrupt. If no physical device is attached to IRQ7 on a slave 8259, you can choose 8259 Spurious (slave) to allow the predefined AMX handler to service the interrupt.

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

ISP Root

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

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

Interrupt Handler

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

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

Handler Language

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

Interrupt Handler Parameter

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

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

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

Spurious Master 8259 Interrupt

If your hardware configuration includes an 8259 Interrupt Controller, then you must be prepared to support the IRQ7 spurious interrupt signal which this controller can generate. If a physical device is attached to IRQ7 on the master 8259, you must provide an Interrupt Service Procedure (ISP) to service the device. That ISP must deal with the spurious IRQ7 interrupt request.

If no physical device is attached to IRQ7 on the master 8259, you can use the AMX handler for the spurious IRQ7 interrupt request. To do so, go to the ISP Definition window, click on the Add button and select 8259 Spurious (master) as the ISP Type. Then edit the default name `---New---` to provide the name you wish to give to your spurious master ISP root. It is up to your application to install the pointer to this ISP root into the Interrupt Descriptor Table using the AMX procedure `cjksispwr()`.

AMX handles the spurious master 8259 interrupt by ignoring the request and dismissing the interrupt with an `IRETD` instruction.

The layout of the window is shown below.
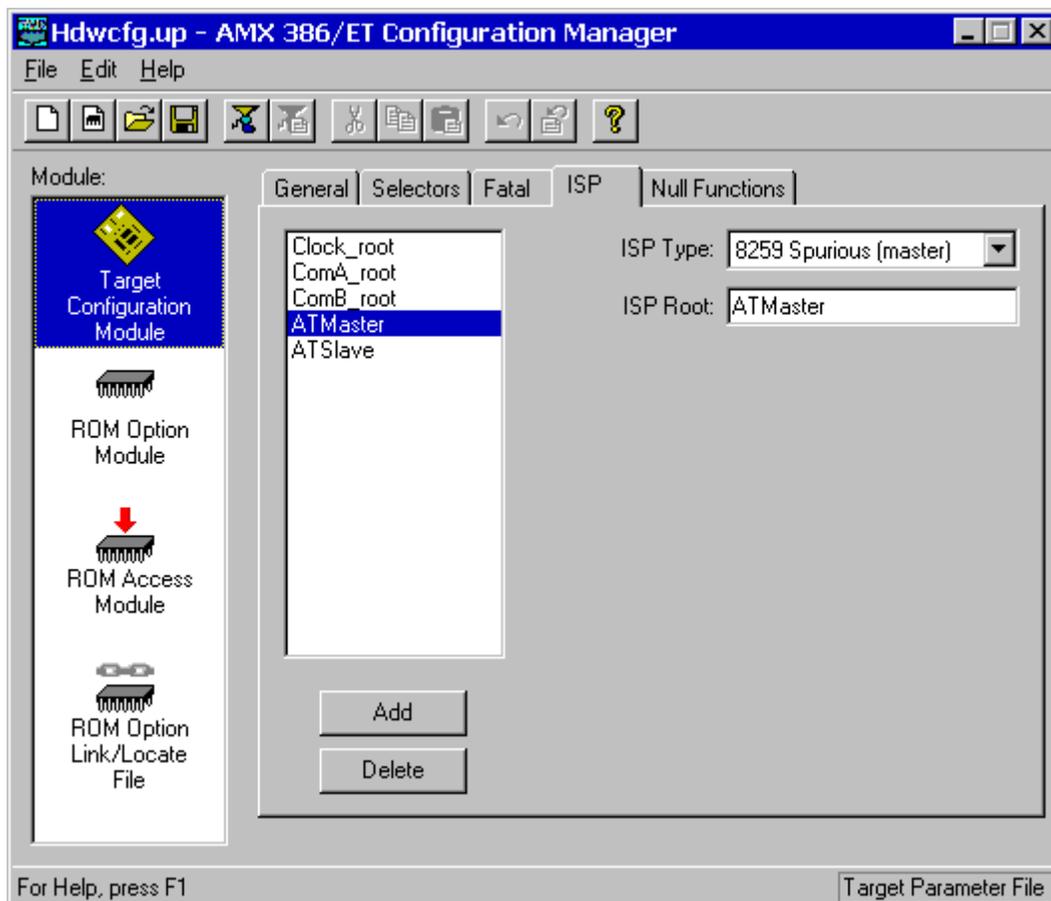
Spurious Slave 8259 Interrupt

If your hardware configuration includes one or more slave 8259 Interrupt Controllers, then you must be prepared to support the IRQ7 spurious interrupt signals which these controllers can generate. A unique spurious interrupt handler is required for each slave 8259 controller. If a physical device is attached to IRQ7 on a slave 8259, you must provide an Interrupt Service Procedure (ISP) to service the device. That ISP must deal with the spurious IRQ7 interrupt request.

If no physical device is attached to IRQ7 on a slave 8259, you can use the AMX handler for the spurious IRQ7 interrupt request. To do so, go to the ISP Definition window, click on the Add button and select 8259 Spurious (slave) as the ISP Type. Then fill in the parameters which describe the ISP handler.

The layout of the window is shown below.
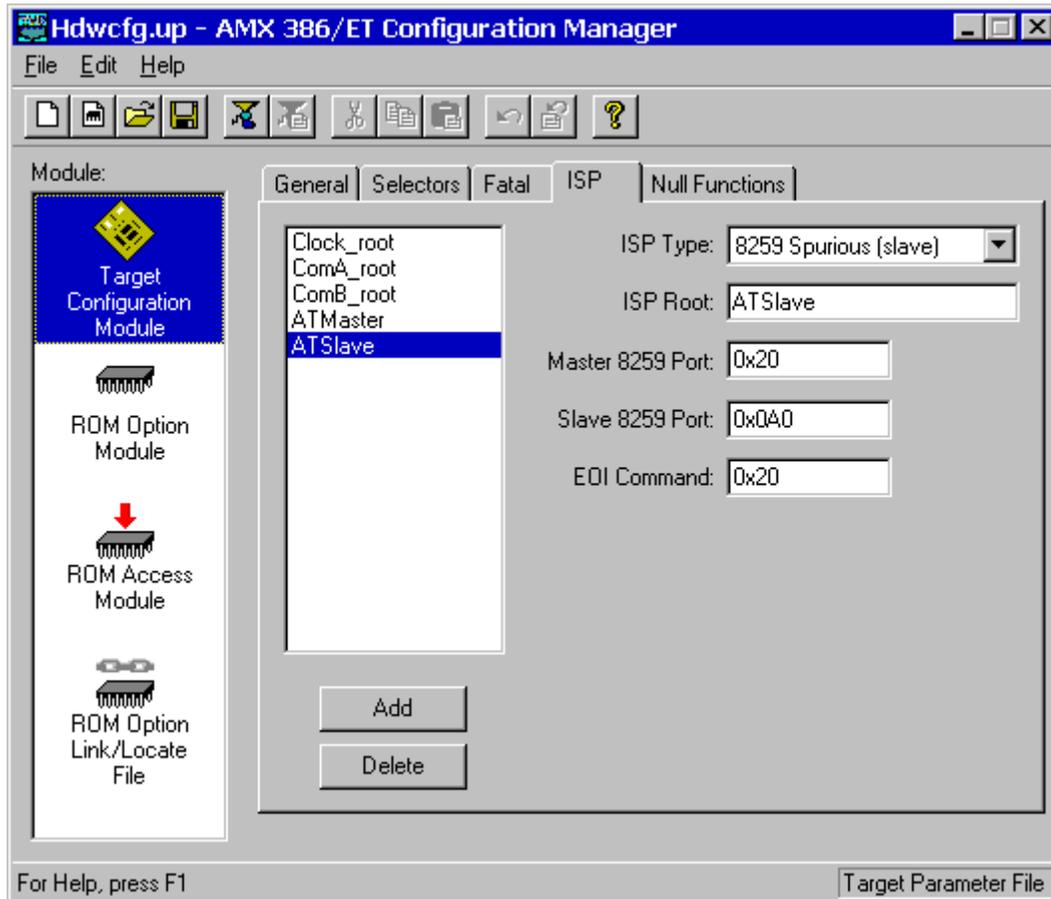
ISP Type

Your ISP for the 8259 IRQ7 spurious interrupt is identified as such by selecting 8259 Spurious (slave) from the pull down list.

ISP Root

Edit the default name ---*New*--- to provide the name you wish to give to your spurious slave ISP root. The ISP root name is used to identify your ISP in the ISP list.

The ISP root is a function created by the AMX Configuration Builder in your Target Configuration Module. The function entry point is declared with a public symbol defined with the name you provide. The name must be unique and must conform to the symbol naming conventions of your assembler. It is up to your application to install the pointer to this ISP root into the Interrupt Descriptor Table using the AMX procedure *cjksispwr()*.

AMX handles the spurious slave 8259 interrupt by ignoring the request, issuing an end-of-interrupt (EOI) command to the master 8259 and dismissing the interrupt with an *IRETD* instruction.

Master and Slave 8259 Port

These parameters specify the 16-bit, hexadecimal values for the device port addresses of the master and slave 8259 interrupt controllers. For PC compatible hardware, the master 8259 port number is *0x20* and the slave 8259 port number is *0xA0*.

EOI Command

This parameter is the 8-bit, hexadecimal value of the command which must be written to the master interrupt controller to dismiss the interrupt request generated by the slave 8259. For PC compatible hardware, the spurious interrupt from the slave 8259 is cleared by writing the non-specific EOI command (*0x20*) to the master 8259 port.

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.
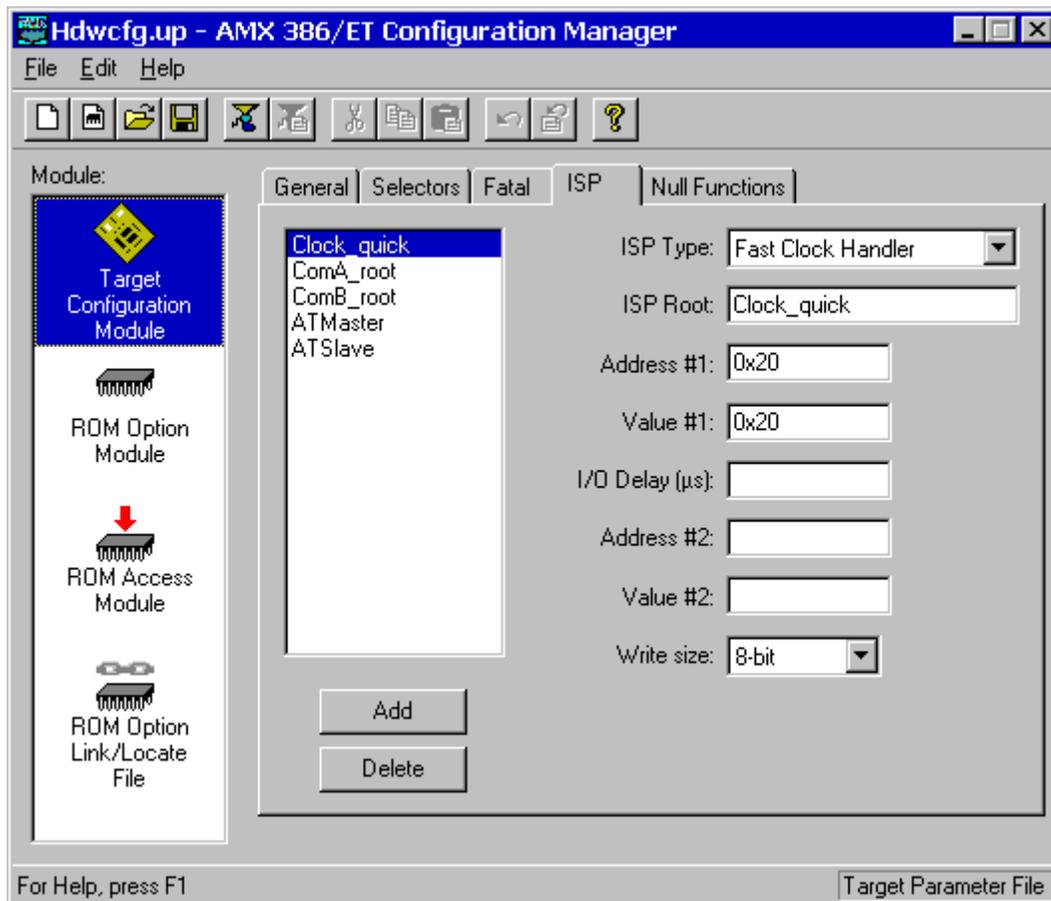
ISP Type

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

Clock Service

Your clock device will be serviced as follows:

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

Address and Value

Each address parameter specifies the 16-bit, hexadecimal value of a device port address which, when referenced with an n-bit value using an i386 I/O instruction, is decoded by your target hardware as a reference to your clock device. Each value parameter is an n-bit, hexadecimal value which must be written to the device port specified by the associated address in order to dismiss the clock interrupt.

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

I/O Delay (μs)

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

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

Write Size

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

4.5 Null Functions

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

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

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

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

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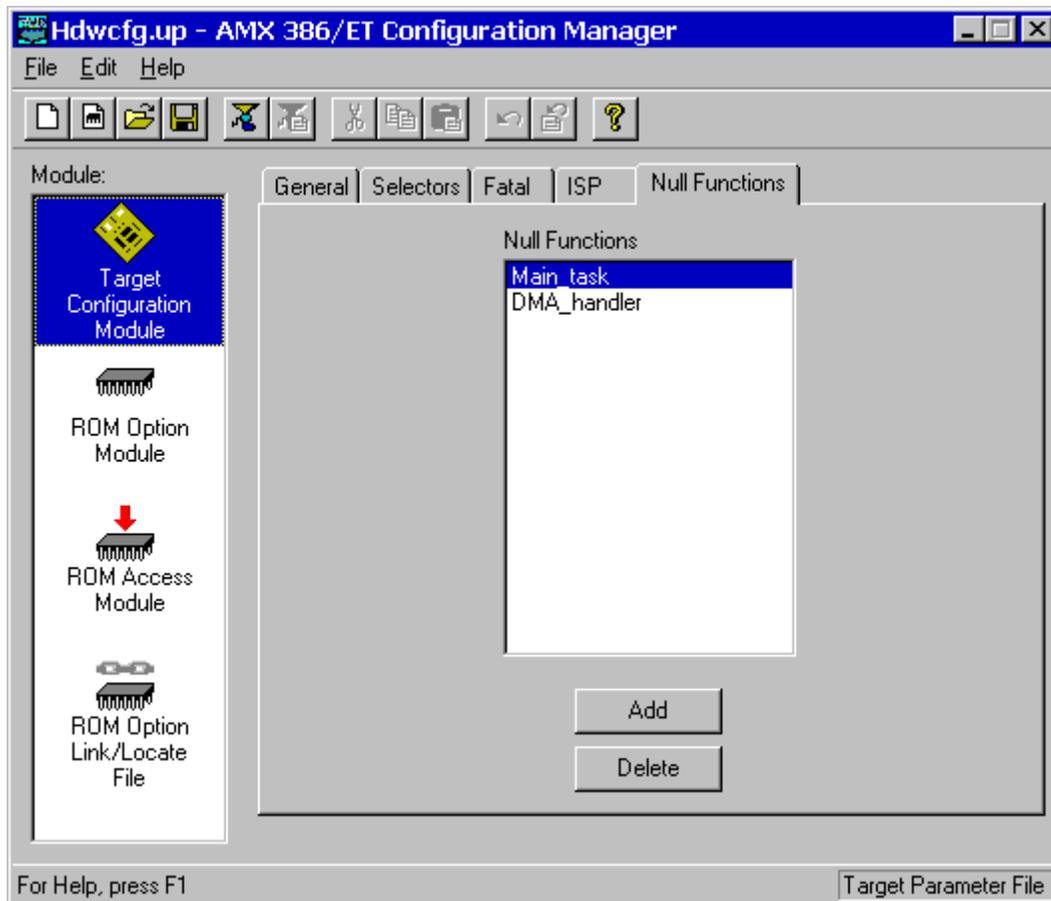
4.6 ROM Option Parameters

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.
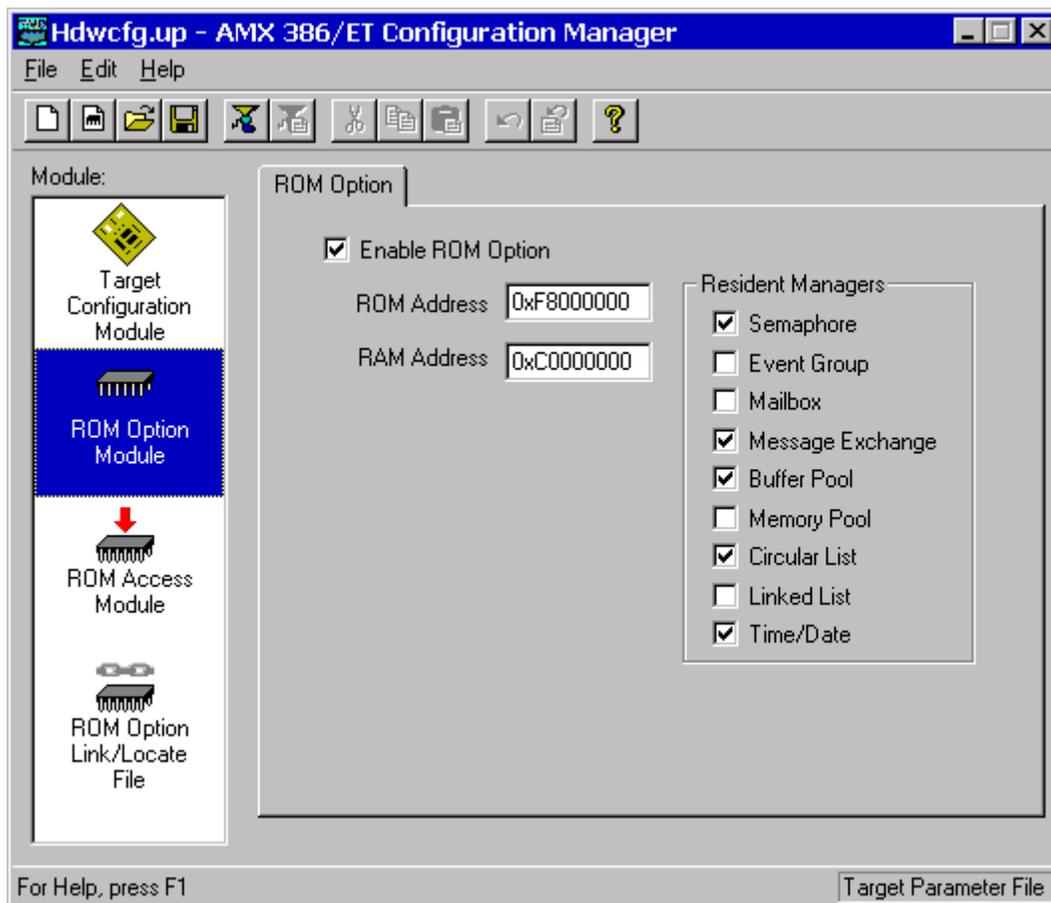
Enable ROM Option

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 Interrupt Descriptor 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 Interrupt Descriptor Table.

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

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

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

Clock Interrupts

A real-time clock used with the i386 processor will interrupt through an interrupt gate defined by an entry in the Interrupt Descriptor Table. The processor will automatically dispatch through the interrupt gate to your clock ISP.

The **clock ISP** consists of an ISP root and an Interrupt Handler. The processor dispatches to the ISP root 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.

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

Clock Shutdown

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

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

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

5.2 Custom Clock Driver

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

The I/O port address of the clock is at *0x03C0*.
The clock interrupt is generated using vector number 125.
The clock interrupt is dismissed by writing bit pattern *0x08* to the clock register at its device address plus 4.

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

```
        PUBLIC   _clockih
_clockih LABEL   NEAR
;
; receives EAX = ISP root parameter = clock device address + 4
;
        MOV     DX,AX           ; DX = clock device address + 4
        MOV     AL,8            ; AL = 8
        OUT     DX,AL           ; Dismiss interrupt
        RET                     ; Return
```

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

ISP Type:	Clock Handler
ISP Root:	*clockroot*
Interrupt Handler:	*_clockih*
Handler Language:	Assembly
Parameter Type:	Value
Parameter:	*0x03C4*

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

ISP Type:	Fast Clock Handler
ISP Root:	*clockroot*
Address #1:	*0x03C4*
Value #1:	*0x08*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	8-bit

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

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

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

   /* Install pointer to clock ISP root into Interrupt Descriptor Table*/
   cjksispwr(125, (CJ_ISPPROC)clockroot);

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

5.3 AMX Clock Drivers

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

5.3.1 PC/AT 8253 (8254) Clock Driver

The AMX clock driver for the Intel 8253 (8254) PIT is ready for use on either a PC/AT or on hardware which incorporates the Intel386EX processor. It is configured to use timer channel 0 operating at 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *AT386\CH8253T.C*.

The 8253 timer generates IRQ0 on the PC/AT master 8259 interrupt controller which is assumed to use the block of 8 descriptors at vector *0x78* in the Interrupt Descriptor Table. Board support module *AT386BRD.C* provides interrupt and clock support services used by this clock driver.

You must compile clock source module *CH8253T.C* and board support module *AT386BRD.C* and link the resulting object modules with the rest of your AMX application.

To use the AMX 8253 clock driver, you must create a fast clock ISP root as described in Chapter 4.4. Use the following parameters in your definition of the clock ISP.

ISP Type:	Fast Clock Handler
ISP Root:	*_ch8253clk*
Address #1:	*0x20*
Value #1:	*0x20*
I/O Delay:	leave blank
Address #2:	leave blank
Value #2:	leave blank
Write Size:	8-bit

Your Target Configuration Module will include a clock ISP root named *_ch8253clk*. The clock driver's initialization procedure will install the pointer to this clock ISP into the Interrupt Descriptor Table. On the PC/AT, the pointer to the clock ISP root is usually installed into the entry for interrupt vector number 8, overwriting any exception handler for the processor double fault exception. However, some debuggers relocate the PC/AT device interrupt vectors 8 to 15 to some other vector, often *0x20* or *0x78*. For this reason, the AMX 8253 clock driver installs the pointer to the ISP root into IDT entry *0x78*.

Clock driver module *CH8253T.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 PC/AT 8253 (8254) Clock Driver

If you wish to use a different 8253 timer channel, change the timer frequency or use a different interrupt vector number, you must edit the definitions in source file *CH8253T.C* and recompile the module. Edit instructions are included in the file.

If you edit this clock driver to use the Intel386EX timer channel 1 or 2, then you must alter the definition of your fast clock ISP as follows.

ISP Type:	Fast Clock Handler
ISP Root:	*_ch8253clk*
Address #1:	*0xA0*
Value #1:	*0x20*
I/O Delay:	*0*
Address #2:	*0x20*
Value #2:	*0x20*
Write Size:	8-bit

If you port the 8253 clock driver to a different hardware platform, you may also have to edit and compile the board support module *AT386BRD.C*.

The board support module *AT386BRD.C* includes a board initialization procedure *chbrdinit* which can be modified to initialize the i386 interrupt system 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.

Appendix A. Target Parameter File Specification

A.1 Target Parameter File Structure

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

```
; AMX Target Parameter File
:
...LAUNCH          PERM,VNA
...HDW             PROC,VMASK,EVTROM,CACHE
...SEL             IDTTYPE,IDTSEL,DGSEL,MEMSEL
...DELAY           CPUFREQ
...M8259           M8259ISP
...S8259           S8259ISP,MPORT,SPORT,EOI-CMD
;
;          Null Functions (optional; one line for each null function)
...NULLFN       FNNAME
;
;          Conforming ISP definitions (one line for each ISP)
...ISPA         ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
...ISPC         ISPROOT,HANDLER,VNUM,PARAM,PARTYPE
;
;          Conforming fast clock ISP (no user code required)
...CLKFAST      CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST16    CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST32    CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
;          or conforming clock ISP (coded in assembly language)
...CLKA         CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
;          or conforming clock ISP (coded in C)
...CLKC         CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
;
;          AMX ROM Option (optional)
...ROMOPT       ROMADR,RAMADR
...ROMSM        ;Semaphore Manager
...ROMEM        ;Event Manager
...ROMMB        ;Mailbox Manager
...ROMMX        ;Message Exchange Manager
...ROMBM        ;Buffer Manager
...ROMMM        ;Memory Manager
...ROMCL        ;Circular List Manager
...ROMLL        ;Linked List Manager
...ROMTD        ;Time/Date Manager
```

Figure A.1-1 AMX Target Parameter File

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

It is the purpose of this appendix to specify all AMX 386/ET directives by defining their keywords and the parameters, if any, which they require.

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

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

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

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

A.2 Target Parameter File Directives

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

```
...LAUNCH          PERM,VNA
```

PERM	0 if the AMX launch is temporary
	1 if the AMX launch is permanent
VNA	0 if the Interrupt Descriptor Table entries are to be alterable
	1 if the Interrupt Descriptor Table entries are NOT to be alterable

You must set *VNA* to 0 to allow AMX or your application to dynamically install ISP pointers into the Interrupt Descriptor Table at run time. If you set *VNA* to 0, you must also set *EVTROM* to 0 in the *...HDW* keyword entry. If you set *VNA* to 1, you must initialize the Interrupt Descriptor Table entries for AMX use as described in Chapter 3.6.

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

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

PROC	Processor identifier
VMASK	= *MMMMMMMMH* = AMX exception vector mask
EVTROM	0 if the Interrupt Descriptor Table is to be in RAM
	1 if the Interrupt Descriptor Table is to be in ROM
CACHE	0 if cache is to be ignored by AMX at launch
	1 if cache is to be enabled by AMX at launch

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

80386	{any of Intel386 DX, SX, CX or EX}
80486	{any of Intel486 DX2, DX or SX}
80586	{Pentium, Pentium II, Pentium Pro}

Note that you can set *PROC* to *80386* to allow your AMX 386/ET application to be used on any of the Intel386, Intel486 or Pentium processors. Be aware that in this case you will NOT have AMX cache support for the Intel486 or Pentium.

Set bit *N* of the *VMASK* Exception Vector Mask for each of the exceptions which are to be serviced by AMX. For example, set this parameter to *00037FFFH* to allow AMX to handle all exceptions. Bits in the mask are defined in Figure 3.2-1. When you are using AMX with a debugger, do not set any of the mask bits for exceptions which the debugger services. For example, a mask of *00000031H* is commonly used with many debuggers.

The *CACHE* parameter can be used to instruct AMX to enable the i386 instruction and data caches when AMX is launched. If the processor or architecture selected with parameter *PROC* has no cache control, set parameter *CACHE* to *0*.

Selector Definitions

The Target Parameter File includes a set of selector definitions.

```
...SEL              IDTTYPE,IDTSEL,DGSEL,MEMSEL
      IDTTYPE       IDT selector type
```
> $IDTTYPE$ IDT selector type
>> = 0 $IDTSEL = xxxxH$ is the actual IDT selector.
>>   The IDT is at address $IDTSEL:0$.
>> = 1 $IDTSEL = xxxxH$ is a selector such that $IDTSEL:0$ references
>>   linear address 0.
>>   The IDT is at address $IDTSEL:IDTADR$ where
>> $IDTADR$ is the linear base address from the $IDTR$ register.
> $DGSEL$ = $xxxxH$ = selector for $DGROUP$ data segment access.
> $MEMSEL$ = $xxxxH$ = selector which can be used to access all of memory
>   using physical addresses.

The $IDTTYPE$ parameter is used to specify the manner in which the address of the Interrupt Descriptor Table is determined.

If $IDTTYPE = 0$, then $IDTSEL$ must be a selector value corresponding to a descriptor in the Global (or Local) Descriptor Table which can be used to access the Interrupt Descriptor Table. The Interrupt Descriptor Table can be accessed by AMX at $IDTSEL:00000000H$.

If $IDTTYPE = 1$, then $IDTSEL$ must be a selector value corresponding to a descriptor in the Global (or Local) Descriptor Table which can be used to access memory beginning at linear address 0. The $LIMIT$ value in the descriptor must be such that the entire Interrupt Descriptor Table lies within the segment. AMX reads the $IDTR$ register to extract the linear base address of the Interrupt Descriptor Table. If the extracted linear address is $IDTADR$, then the Interrupt Descriptor Table can be accessed by AMX at $IDTSEL:IDTADR$.

The value for $IDTSEL$ must be derived from your link and locate tool. Most tools give you the ability to identify or allocate specific selectors. If you do not know the selector value, set parameter $IDTSEL$ to 0.

$DGSEL$ is the run-time selector value corresponding to a descriptor in the Global (or Local) Descriptor Table which is used to access the read/write data segment commonly described as a member of group $DGROUP$. Your AMX application is assumed to execute with selector $DGSEL$ in segment register DS.

$MEMSEL$ is the run-time selector which can be used to access all of physical memory. Memory at physical (absolute) address 0 is referenced as $MEMSEL:0$. For most toolsets which use the flat memory model, $MEMSEL$ is the same as $DGSEL$.

Device I/O Delay

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

```
...DELAY          CPUFREQ

     CPUFREQ      i386 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.

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.

Master and Slave 8259 Spurious Interrupt ISPs

AMX 386/ET includes default spurious interrupt handlers for both master and slave 8259 interrupt controllers. To include these handlers in your AMX system, add the following statements to your AMX 386/ET Target Parameter File.

```
...M8259          M8259ISP
...S8259          S8259ISP,MPORT,SPORT,EOI-CMD

     M8259ISP     is the name to be assigned to your master 8259 ISP root.
     S8259ISP     is the name to be assigned to your slave 8259 ISP root.
     MPORT        is the master 8259 device port address.
     SPORT        is the slave 8259 device port address.
     EOI-CMD      is the 8259 command byte to be issued to the master 8259
                  to clear a spurious interrupt occurring on the slave 8259.
```

If IRQ7 is used on the master 8259 interrupt controller, omit the `...M8259` directive. If you have no slave 8259 interrupt controller, omit the `...S8259` directive. If you have more than one slave controller, you will need a separate `...S8259` directive for each slave. If IRQ7 is used on a slave 8259 interrupt controller, omit the `...S8259` directive for that slave.

When your AMX system is launched, you must update the Interrupt Descriptor Table so that the entries for the master and slave 8259 IRQ7 interrupts reference these ISP roots in your Target Configuration Module.

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 *...ISPC* if its Interrupt Handler is coded in C or *...ISPA* if its Interrupt Handler is coded in assembly language.

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

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

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

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

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

VNUM defines the interrupt vector number which you have assigned to the device. *VNUM* is 0 to 255. Note that all vector numbers in the range 0 to 31 are reserved by Intel.

If *VNUM* is 0 to 255, AMX will automatically install the pointer to the ISP root *ISPROOT* into the corresponding entry in the Interrupt Descriptor 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 Interrupt Descriptor Table using AMX procedure *cjksispwr*, *cjksidtwr* or *cjksidtx*.

Note

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

—▯KADAK—

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.

```
...CLKC          CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
...CLKA          CLKROOT,CLKHAND,VNUM,PARAM,PARTYPE
...CLKFAST       CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST16     CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
...CLKFAST32     CLKROOT,CLKADR,CLKCMD,CLKADR2,CLKCMD2,IODELAY,VNUM
```

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

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

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

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

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

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

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

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

CLKROOT	Name of the clock ISP root entry point
CLKADR	16-bit numeric device port address
CLKCMD	8-bit numeric command
CLKADR2	16-bit numeric secondary device port address
CLKCMD2	8-bit numeric secondary command
IODELAY	Delay (μs) required between I/O commands
VNUM	Interrupt vector number assigned to the clock device

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

The clock ISP root will dismiss the clock interrupt by writing the 8-bit value *CLKCMD* to the 16-bit device port *CLKADR*. If parameter *CLKADR2* is present in the *...CLKFAST* directive, the clock ISP root will then write the 8-bit value to the 16-bit device port *CLKADR2*. If parameter *CLKADR2* is present, parameter *CLKCMD2* must also be present. If this second device I/O command is not required, leave both *CLKCMD2* and *CLKADR2* blank (empty).

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

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

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

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

—⊞KADAK—

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 used to determine the ROM address at which the AMX ROM image is to be located. This address is dictated by you according to your hardware requirements. If *CSEL* is the run-time code selector for your AMX application, then *CSEL:ROMADR* specifies the run-time selector and offset which can be used to access the AMX code segment.

Parameter *RAMADR* is used to determine the RAM address of a block of 32 bytes reserved for use by AMX. If *DGSEL* is the run-time *DGROUP* data selector for your application (see keyword *...SEL*), then *DGSEL:RAMADR* specifies the run-time selector and offset which can be used to access this private AMX data segment.

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

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

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

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

This page left blank intentionally.

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.ASM* from the AMX 386/ET Target Configuration Template File *CJ812HDW.CT* and a Target Parameter File called *HDWCFG.UP* is as follows:

```
CJ812CG HDWCFG.UP CJ812HDW.CT HDWCFG.ASM
```

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

This page left blank intentionally.

Appendix B. AMX 386/ET Service Procedures

B.1 Summary of Services

AMX 386/ET provides a collection of target dependent AMX service procedures for use with the i386 processor and compatibles and the C compilers which support them. These procedures are summarized below.

Interrupt Control (class *ksi*)

cjksidtm	Make an interrupt gate description
cjksidtrd	Read an entry from the Interrupt Descriptor Table
cjksidtwr	Write an entry into the Interrupt Descriptor Table
cjksidtx	Exchange an entry in the Interrupt Descriptor Table
cjksispwr	Install an ISP pointer as an interrupt gate in an entry in the Interrupt Descriptor Table
cjksitrap	Install a task trap handler

Processor and C Interface Procedures (class *cf*)

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

cjcfccsetup	Setup C environment
cjcfdi	Disable interrupts
cjcfei	Enable interrupts
cjcfflagrd	Read the processor flags register (*EFLAGS*)
cjcfflagwr	Write to the processor flags register (*EFLAGS*)
cjcfhwdelay	Delay *n* microseconds
cjcfhwbcache	Flush and enable/disable data and instruction caches
cjcfhwdcache	Flush and enable/disable data cache
cjcfhwicache	Flush and enable/disable instruction cache
cjcfinp8	Read an 8-bit input port
cjcfinp16	Read a 16-bit input port
cjcfinp32	Read a 32-bit input port
cjcfjlong	Long jump to a mark set by *cjcfjset*
cjcfjset	Set a mark for a subsequent long jump by *cjcfjlong*
cjcfmcopy	Copy a block of memory
cjcfmset	Set (fill) a block of memory
cjcfoutp8	Write an 8-bit value to an output port
cjcfoutp16	Write a 16-bit value to an output port
cjcfoutp32	Write a 32-bit value to an output port
cjcfstkjmp	Switch stacks and jump to a new procedure
cjcftag	Convert a string to an AMX tag value
cjcfvol8	Read a volatile 8-bit variable
cjcfvol16	Read a volatile 16-bit variable
cjcfvol32	Read a volatile 32-bit variable
cjcfvolpntr	Read a volatile pointer variable

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

cjcfregld	Load i386 registers from a register array
cjcfregst	Store i386 registers into a register array
cjcfsint	Generate a software interrupt

B.2 Service Procedures

A description of all processor dependent AMX 386/ET 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   ■ Restored
                     (Not in ISP)

D E R   Effect on Interrupts

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

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

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

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

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

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

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

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

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

Purpose	**Setup C Environment**

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

Setup   Prototype is in file *CJZZZIF.H*.
#include "CJZZZ.H"
void CJ_CCPP cjcfccsetup(void);

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

Interrupts   □ Disabled   □ Enabled   □ Restored

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

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

Purpose	**Disable or Enable Interrupts**

Used by   ■ Task   ■ ISP   ■ Timer Procedure   ■ Restart Procedure   ■ Exit Procedure
(cjcfdi only)

Setup   Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.
```
#include "CJZZZ.H"
void CJ_CCPP cjcfdi(void);
void CJ_CCPP cjcfei(void);
```

Description Tasks can use *cjcfdi* to briefly disable all sources of interrupt. Immediately thereafter the task can use *cjcfei* to enable all sources of interrupt again.

Interrupts   ■ Disabled by *cjcfdi*   ■ Enabled by *cjcfei*

Returns   Nothing

The interrupt enable flag in the flags register (*EFLAGS*) is reset to 0 to disable interrupts or set to 1 to enable interrupts.

Restrictions ISPs must not use *cjcfei* unless nested interrupts are supported in your application by external hardware such as the Intel 8259 Interrupt Controller.

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

See Also   *cjcfflagrd, cjcfflagwr*

Purpose　　**Read or Write Processor Flags Register**

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

Setup　　　Prototype in file `CJZZZTF.H` or macro in file `CJZZZCC.H`.
```
#include "CJZZZ.H"
CJ_TYFLAGS CJ_CCPP cjcfflagrd(void);
void CJ_CCPP cjcfflagwr(CJ_TYFLAGS flags);
```

Description　`Cjcfflagrd` returns the actual state of the processor flags register (`EFLAGS`).

`Cjcfflagwr` updates the processor flags register (`EFLAGS`) by writing the parameter `flags` to the register.

Use `cjcfflagrd` to read the state of the processor flags 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 flags register.
`Cjcfflagwr` returns nothing.

`Cjcfflagwr` unconditionally copies `flags` into the processor flags register, thereby enabling or disabling interrupts. Since no validation of `flags` is performed, caution in the use of `cjcfflagwr` is advised.

See Also　　`cjcfdi, cjcfei`

Purpose   **Delay n Microseconds**

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

Setup   Prototype is in file *CJZZZIF.H*.
                 #include "CJZZZ.H"
                 void CJ_CCPP cjcfhwdelay(int n);

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

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

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

Interrupts   □ Disabled   □ Enabled   □ Restored

Returns   Nothing

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

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

Purpose   **Flush and Enable/Disable Caches**

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

Setup   Prototype in file `CJZZZIF.H`.
                 ```
                 #include "CJZZZ.H"
                 void CJ_CCPP cjcfhwbcache(int operation);
                 void CJ_CCPP cjcfhwdcache(int operation);
                 void CJ_CCPP cjcfhwicache(int operation);
                 ```

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

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

Interrupts   ■ Disabled   □ Enabled   ■ Restored

Returns   Nothing
                 `Cjcfhwbcache` flushes and disables (or enables) both the data and
                 instruction caches.

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

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

Note   These procedures can be called even if your Target Parameter File
                 indicates that you are targeting an 80x86 processor with no cache or only
                 one kind of cache. In such cases, the procedures only affect the caches
                 which exist. If a single cache is used for both instruction and data
                 caching, only procedure `cjcfhwbcache` should be used since both caches
                 will always be affected.

Restrictions ISPs and Timer Procedures must not use these procedures. Since
                 interrupts are disabled while the caches are flushed, use caution when
                 calling these procedures or system performance will be degraded,
                 especially if the cache sizes are large.

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

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

Setup   Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.
```
#include "CJZZZ.H"
CJ_T8 CJ_CCPP cjcfinp8(int port);
CJ_T16 CJ_CCPP cjcfinp16(int port);
CJ_T32 CJ_CCPP cjcfinp32(int port);
```

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

Interrupts   □ Disabled   □ Enabled   □ Restored

Returns   *Cjcfinp8* returns an 8-bit signed value.
Cjcfinp16 returns a 16-bit signed value.
Cjcfinp32 returns a 32-bit signed value.

Example
```
#include "CJZZZ.H"

#define CONSTAT (0x12D)      /* Console status register   */
#define CONDATA (0x12F)      /* Console data register     */


void CJ_CCPP conout(char ch) {

                             /* Wait for ready            */
  while ( (cjcfinp8(CONSTAT) & 0x80) == 0 )
        ;

                             /* Write character           */
  cjcfoutp8(CONDATA, (CJ_T32)ch);
  }
```

See Also   *cjcfoutp8, cjcfoutp16, cjcfoutp32*

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

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

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

Setup   Prototype is in file *CJZZZTF.H*.
                   ```
                   #include "CJZZZ.H"
                   void CJ_CCPP cjcfjlong(struct cjxjbuf *jbuf, int value);
                   int CJ_CCPP cjcfjset(struct cjxjbuf *jbuf);
                   ```

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

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

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

Interrupts   □ Disabled   ■ Enabled   □ Restored

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

                   There is no return from *cjcfjlong*.

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

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

Example

```
#include "CJZZZ.H"

void CJ_CCPP dowork(struct cjxjbuf *jbp);

static struct cjxjbuf jumpbuffer;

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

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

void CJ_CCPP taskbody(void) {

  if (cjcfjset(&jumpbuffer) == 0)

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

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


void CJ_CCPP dowork(struct cjxjbuf *jbp) {

  /* Do work                                         */

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

See Also   `cjcfstkjmp`

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

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

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

Setup　　Prototype is in file *CJZZZTF.H*.
```
#include "CJZZZ.H"
void CJ_CCPP cjcfmcopy(int *sourcep, int *destp,
                       unsigned int size);
void CJ_CCPP cjcfmset(int *mempntr,
                       unsigned int size, int pattern);
```

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

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

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

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

Interrupts　□ Disabled　　□ Enabled　　□ Restored

Returns　Nothing

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

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

Example
```
#include "CJZZZ.H"

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


void CJ_CCPP blocksetcopy(int pattern) {

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

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

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

Setup　　Prototype in file *CJZZZTF.H* or macro in file *CJZZZCC.H*.
```
#include "CJZZZ.H"
void CJ_CCPP cjcfoutp8(int port, CJ_T32 data);
void CJ_CCPP cjcfoutp16(int port, CJ_T32 data);
void CJ_CCPP cjcfoutp32(int port, CJ_T32 data);
```

Description　*port* is the port number 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
Cjcfoutp8 outputs the least significant 8 bits of *data* to the port.
Cjcfoutp16 outputs the least significant 16 bits of *data* to the port.
Cjcfoutp32 outputs the full 32 bits of *data* to the port.

Example
```
#include "CJZZZ.H"

#define CONSTAT (0x12D)     /* Console status register  */
#define CONDATA (0x12F)     /* Console data register    */


void CJ_CCPP conout(char ch) {

                            /* Wait for ready           */
  while ( (cjcfinp8(CONSTAT) & 0x80) == 0 )
        ;

                            /* Write character          */
  cjcfoutp8(CONDATA, (CJ_T32)ch);
  }
```

See Also　*cjcfinp8, cjcfinp16, cjcfinp32*

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

This procedure is provided for AMX portability.

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

Setup   Prototype is in file *CJZZZTF.H*.
```
#include "CJZZZ.H"
void CJ_CCPP cjcfstkjmp(void *vp, void *stackp,
                        CJ_VPPROC procp);
```

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

stackp is a pointer to a properly aligned block of memory for use as a stack. For the 80x86 family, the stack must be 16-bit word aligned. For most 80386, 80486 and Pentium processors, performance will be improved if the stack is 32-bit long word aligned.

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*

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

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

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

Setup   Prototype in file `CJZZZTF.H` or macro in file `CJZZZCC.H`.
```
#include "CJZZZ.H"
CJ_T8 CJ_CCPP cjcfvol8(void *varp);
CJ_T16 CJ_CCPP cjcfvol16(void *varp);
CJ_T32 CJ_CCPP cjcfvol32(void *varp);
void * CJ_CCPP cjcfvolpntr(void *pntrp);
```

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

`pntrp` is a pointer to a pointer variable.

Interrupts ☐ Disabled   ☐ Enabled   ☐ Restored

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

Example
```
#include "CJZZZ.H"

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


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

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

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

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

This page left blank intentionally.

Purpose   **Make an Unpacked Interrupt Descriptor**

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

Setup   Prototype is in file *CJZZZIF.H*.
```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjksidtm(CJ_ISPPROC ispproc,
                          struct cjxidts *gdp);
```

Description   *ispproc* is a pointer to the Interrupt Service Procedure to be installed in
                  the interrupt descriptor.

                  gdp is a pointer to storage for an unpacked interrupt descriptor. Structure
                  cjxidts is defined in file *CJZZZKT.H* as follows:

```
struct cjxidts {
    CJ_T32U      xidofs;      /* Procedure offset         */
    CJ_T16U      xidsel;      /* Procedure selector       */
    CJ_T16U      xidtype;     /* Gate type, access rights  */
    };
```

Interrupts   □ Disabled   □ Enabled   □ Restored

Returns   Error status is returned.
                  CJ_EROK   Call successful.
                  *gdp contains the unpacked interrupt descriptor defining an interrupt
                  gate marked as present with privilege level 0.

                  The unpacked interrupt descriptor includes a *FAR* pointer to Interrupt
                  Service Procedure *ispproc*. The pointer is constructed using the AMX
                  code selector and the offset of procedure *ispproc*.

                  The unpacked interrupt descriptor can be packed and installed in the
                  Interrupt Descriptor Table using procedure *cjksidtwr* or *cjksidtx*.

                  Errors returned:
                  None

See Also   *cjksidtwr, cjksidtx*

Purpose   **Read from the Interrupt Descriptor 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 cjksidtrd(int vector, struct cjxidts *gdp);
```

Description   *vector* is the processor vector number (0 to 255).

gdp is a pointer to storage for an unpacked copy of the descriptor retrieved
   from the specified entry in the Interrupt Descriptor Table. Structure
   cjxidts is defined in file *CJZZZKT.H* as follows:

```
struct cjxidts {
   CJ_T32U       xidofs;     /* Procedure offset           */
   CJ_T16U       xidsel;     /* Procedure selector         */
   CJ_T16U       xidtype;    /* Gate type, access rights    */
   };
```

Interrupts   ■ Disabled   □ Enabled   ■ Restored

Returns   Error status is returned.
   CJ_EROK   Call successful.
   *gdp contains the unpacked descriptor retrieved from Interrupt
   Descriptor Table entry number *vector*. The unpacked descriptor
   includes the Interrupt Service Procedure pointer (or exception service
   procedure pointer) together with the gate type and access rights.

   Errors returned:
   None

See Also   *cjksidtm, cjksidtwr, cjksidtx*

Purpose   **Write to the Interrupt Descriptor 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 cjksidtwr(int vector, struct cjxidts *gdp);
```

Description   `vector` is the processor vector number (0 to 255).

               `gdp` is a pointer to an unpacked descriptor to be packed and installed into the specified entry in the Interrupt Descriptor Table. Structure `cjxidts` is defined in file `CJZZZKT.H` as follows:

```
struct cjxidts {
    CJ_T32U      xidofs;      /* Procedure offset        */
    CJ_T16U      xidsel;      /* Procedure selector      */
    CJ_T16U      xidtype;     /* Gate type, access rights */
    };
```

               The unpacked descriptor includes a `FAR` pointer to the Interrupt Service Procedure (or exception service procedure). The gate type and access rights must be one of the following:

               `CJ_IDTINT`   Interrupt gate, present, privilege level 0
               `CJ_IDTTRAP`   Trap gate, present, privilege level 0

Interrupts   ■ Disabled   □ Enabled   ■ Restored

Returns   Error status is returned.
               `CJ_EROK`   Call successful.

               Errors returned:
             `CJ_ERNOACCESS` Interrupt Descriptor Table is not accessible. AMX was launched with access to the Interrupt Descriptor Table denied.

Restrictions You must NOT use this procedure to alter the task trap vectors (`vector = CJ_PRVNZD`, `CJ_PRVNBD` or `CJ_PRVNOV`). Use `cjksitrap` for that purpose.

Example

```
#include "CJZZZ.H"

#define INTNUM 212          /* Device interrupt number  */

void CJ_CCPP isproot(void);  /* ISP root                */


void CJ_CCPP ispRR(void) {   /* ISP Restart Procedure   */
  struct cjxidts ispgate;

  /* Make an unpacked interrupt descriptor            */
  cjksidtm((CJ_ISPPROC)isproot, &ispgate);

  /* Pack and install the descriptor in the IDT       */
  cjksidtwr(INTNUM, &ispgate);
  }
```

See Also   `cjksidtm, cjksidtrd, cjksidtx`

Purpose   **Exchange an Entry in the Interrupt Descriptor 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 cjksidtx(int vector,
                          struct cjxidts *newgdp,
                          struct cjxidts *oldgdp);
```

Description   *vector* is the processor vector number (0 to 255).

newgdp is a pointer to an unpacked descriptor to be packed and installed
into the specified entry in the Interrupt Descriptor Table. Structure
cjxidts is defined in file *CJZZZKT.H* as follows:

```
struct cjxidts {
    CJ_T32U      xidofs;     /* Procedure offset          */
    CJ_T16U      xidsel;     /* Procedure selector        */
    CJ_T16U      xidtype;    /* Gate type, access rights  */
    };
```

The unpacked descriptor includes a *FAR* pointer to the Interrupt Service
Procedure (or exception service procedure). The gate type and access
rights must be one of the following:

CJ_IDTINT   Interrupt gate, present, privilege level 0
CJ_IDTTRAP   Trap gate, present, privilege level 0

oldgdp is a pointer to storage for an unpacked copy of the previous
descriptor retrieved from the specified entry in the Interrupt Descriptor
Table.

Interrupts   ■ Disabled   □ Enabled   ■ Restored

Returns   Error status is returned.
   CJ_EROK   Call successful.
   *oldgdp contains the unpacked copy of the previous descriptor
   retrieved from Interrupt Descriptor Table entry number *vector*. The
   unpacked descriptor includes the Interrupt Service Procedure pointer
   (or exception service procedure pointer) together with the gate type and
   access rights.

   Errors returned:
     For all errors, **oldgdp* is undefined on return.
     CJ_ERNOACCESS Interrupt Descriptor Table is not accessible.
               AMX was launched with access to the
               Interrupt Descriptor Table denied.

Restrictions You must NOT use this procedure to alter the task trap vectors (`vector = CJ_PRVNZD`, `CJ_PRVNBD` or `CJ_PRVNOV`). Use `cjksitrap` for that purpose.

Example
```
#include "CJZZZ.H"

#define INTNUM 212          /* Device interrupt number  */

void CJ_CCPP isproot(void); /* ISP root                 */

                            /* Old descriptor           */
static struct cjxidts oldgate;


void CJ_CCPP ispRR(void) {  /* ISP Restart Procedure    */
  struct cjxidts newgate;

  /* Make an unpacked interrupt descriptor              */
  cjksidtm((CJ_ISPPROC)isproot, &newgate);

  /* Save old descriptor and install new one in the IDT */
  cjksidtx(INTNUM, &newgate, &oldgate);
  }


void CJ_CCPP ispEX(void) {  /* ISP Exit Procedure       */

  /* Install original descriptor in IDT                 */
  cjksidtwr(INTNUM, &oldgate);
  }
```

See Also   `cjksidtm, cjksidtrd, cjksidtwr`

Purpose	**Install an ISP in the Interrupt Descriptor 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 cjksispwr(int vector, CJ_ISPPROC newproc);

Description   *vector* is the processor vector number (0 to 255).

newproc is a pointer to the new Interrupt Service Procedure.

Interrupts   ■ Disabled   □ Enabled   ■ Restored

Returns   Error status is returned.
　　CJ_EROK   Call successful.
　　The Interrupt Service Procedure pointer *newproc* is installed into
　　Interrupt Descriptor Table entry number *vector* as an interrupt gate
　　marked as present with privilege level 0.

Errors returned:
　　CJ_ERNOACCESS   Interrupt Descriptor Table is not accessible.
　　　　　　　　　　　AMX was launched with access to the
　　　　　　　　　　　Interrupt Descriptor Table denied.

Restrictions   You must NOT use this procedure to alter the task trap vectors (*vector = CJ_PRVNZD*, *CJ_PRVNBD* or *CJ_PRVNOV*). Use *cjksitrap* for that purpose.

See Also   *cjksidtrd, cjksidtx*

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 processor vector number which identifies the particular error trap.

CJ_PRVNZD	Zero divide trap
CJ_PRVNBD	*BOUND* (bounds check) instruction trap
CJ_PRVNOV	*INTO* (overflow test) instruction 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.

Appendix C. AMX 386/ET 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.

CJ722ROP.LKT	AMX ROM Option toolset dependent Link Specification Template
CJ722ROP.CT	AMX ROM Option Template
CJ722RAC.CT	AMX ROM Access Template

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

Creating an AMX ROM

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

The Configuration Generator combines the information in your Target Parameter File with the ROM Option Template file *CJ722ROP.CT* to produce an assembly language ROM Option Module *CJ722ROP.ASM*.

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:

```
CJ722CG HDWCFG.UP CJ722ROP.CT CJ722ROP.ASM
```

The ROM Option Module *CJ722ROP.ASM* is then assembled in exactly the same manner as your Target Configuration Module *HDWCFG.ASM* 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 *.ASM* 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 `CJ722RAC.CT` to produce an assembly language ROM Access Module `CJ722RAC.ASM`.

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:

```
CJ722CG HDWCFG.UP CJ722RAC.CT CJ722RAC.ASM
```

The ROM Access Module `CJ722RAC.ASM` is then assembled in exactly the same manner as your Target Configuration Module `HDWCFG.ASM` 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 `CJ722RAC.OBJ` 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 `CJ722.LIB`.

Note that you must still include the AMX Library `CJ722.LIB` 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 `.OBJ` and `.LIB` 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. i386 Bootstrap Initialization

D.1 Introduction

AMX 386/ET is designed to execute with the i386 processor in protected mode. However, following a hardware *RESET*, the i386 processor is in real mode. Additional initialization must be performed by software to enter protected mode before AMX can be launched.

The minimal protected mode requirements for an AMX launch are described in Chapter 1.5. If you are using a debugger and/or a ROM monitor in your target i386 hardware to load and execute your AMX 386/ET application, then these minimal requirements will be met. The debugger or ROM monitor prepares the protected mode environment when it loads your program.

In an embedded ROM system, the initialization for protected mode operation is typically performed by the real mode bootstrap code vectored to by the hardware reset.

This chapter describes the minimal initialization needed to support AMX and provides example bootstrap code suitable for use in a ROMed AMX system.

> Note
>
> You should only have to use the AMX 386/ET bootstrap code if none is provided with the development tools which you are using.

D.2 Protected Mode Initialization

The minimal requirements for the AMX 386/ET protected mode environment are as follows. The GDT register must reference a valid Global Descriptor Table (GDT) containing a readable code segment descriptor and a writable data segment descriptor. The IDT register must reference a valid Interrupt Descriptor Table (IDT). Use of the Local Descriptor Table (LDT) or i386 memory paging or debugging registers is not required. AMX will not prevent you from using them if you so desire. Thus the *TR*, *CR1..CR3* and *DR0..DR7* registers may be left uninitialized.

The general initialization steps are as follows.

> Disable interrupts and reset all interrupt generating devices.
> Create a Global Descriptor Table.
> Load the *GDTR* and switch to protected mode.
> Load the *IDTR*.
> Launch AMX.
> If the IDT is in RAM, initialize the IDT interrupt gates for your
> interrupt generating devices.

To manage the transition from real mode to protected mode AMX provides the following services in source file *CJ722BSU.ASM*:

> *CJ_PMBLD*   Create a Global Descriptor Table
> *CJ_PMSW*   Load *GDTR* and switch to protected mode

To setup the IDT register (*IDTR*), use AMX library procedure *CJ_PMIDTS*. See source module *CJ722KKA.ASM* for the assembly language calling sequence.

Creating a Global Descriptor Table

Some linkers/locators will create the necessary GDT and IDT data structures for you. Alternatively, your bootstrap code can create its own GDT using AMX procedure *CJ_PMBLD*.

CJ_PMBLD is a real mode *NEAR* procedure that creates a Global Descriptor Table in RAM from a simple Segment Description Table.

The calling sequence for *CJ_PMBLD* is as follows:

> *ES:SI* = A(Segment Description Table)
> *DS:DI* = A(RAM storage for GDT)
> *CX*　　= size of GDT (bytes)
> *CALL　CJ_PMBLD*
> *EAX* is altered.
> All other registers are unaltered.
> Interrupts are not touched.

The Segment Description Table defines the segments for which descriptors must be built in the GDT. A minimum of two segments must be defined, one for code and one for data.

The RAM storage allocated for use as the GDT must contain 8 bytes for each segment descriptor. Entries in the GDT are referenced by an index (*0* to *n-1*). Entry *0* is unused but must be present. Entry *1* is reserved for use by AMX. The remaining entries will be used according to the specifications in your Segment Description Table.

Note that the GDT must be large enough to hold entries for all of the segments defined in your Segment Description Table. This may imply that some entries in the GDT are unused and hence not initialized.

Segment Description Table

The Segment Description Table defines the segments for which descriptions will be built in the GDT.

At least two segments must be defined. One segment must be a readable, executable code segment. The other segment must be a read/write data segment. These two segments correspond to the *CS* and *DS* runtime selectors which you have chosen for your Flat (Small) model AMX system.

Other segments may be defined as well if required by your application.

The Segment Description Table consists of an array of segment descriptions followed by a byte containing 0 to terminate the array. Figure D.2-1 illustrates a typical Segment Description Table.

In Figure D.2-1, parameters shown as *<xxxxxx>* must be defined by you.

```
; AMX Segment Description Table
;
SEGDT LABEL BYTE
;
; Define your segments here
;
;      :
;      :
;
; Define each segment as follows:
;
      DB    <type>                  ; segment type
      DB    0                       ; reserved
      DW    <index>*8               ; GDT entry (selector)
      DD    <baseadr>               ; linear base address
      DD    <limit>                 ; segment limit = size-1 (bytes)
;
      DB    0                       ; end of Segment Description Table
;
```

Figure D.2-1 Segment Description Table

Each segment is described with a single entry in the Segment Description Table. The following parameters must be included in the definition of each segment.

```
DB      <type>              ; segment type
DB      0                   ; reserved
```

The segment type must be one of the following constants provided in the definition file *CJ722K.DEF*.

K_PMRODS   Read only data segment
K_PMRWDS   Read/write data segment
K_PMEOCS   Execute only code segment
K_PMERCS   Readable and executable code segment

```
DW      <index>*8           ; GDT entry (selector)
```

This parameter selects the entry in the GDT in which the segment descriptor is to be built. *Index* must be greater than or equal to 2. Note that *<index>*8* is therefore the privilege level 0 selector value (*10H*, *18H*, *20H* etc.) to be used to reference the segment.

```
DD      <baseadr>           ; linear base address
```

This parameter defines the 32-bit linear address of the start of the segment.

```
DD      <limit>             ; segment limit = size-1 (bytes)
```

This parameter defines the size of the segment. Note that the *limit* is specified as *size-1* bytes. Note that if the limit exceeds 1MB, the limit will be rounded up to the nearest 4KB boundary because of the way the i386 performs limit checking.

D.3 Switching to Protected Mode

Once a valid Global Descriptor Table (GDT) is defined, you can switch to protected mode and execute your protected mode startup code by calling AMX procedure *CJ_PMSW*.

The calling sequence for *CJ_PMSW* is as follows:

> Interrupts must be disabled.
> *DS:DI* = A(GDT) with selector *8* providing a data alias for the GDT
> *ES:SI* = A(Application Initialization Block)
> *SS:SP* = Stack pointer with at least 8 bytes of stack free for use
> *JMP   CJ_PMSW*

The GDT must be valid before *CJ_PMSW* is called. In particular, you must ensure that the GDT entry at index *1* (selector value *8*) contains a valid data alias for the GDT itself. If you used AMX procedure *CJ_PMBLD* to create your GDT, this condition will be met.

The Application Initialization Block defines the start address of your protected mode startup code and the initial stack to be used on entry to it. The block must be structured as follows.

```
DD    OFFSET <stack>           ; protected mode ESP
DW    <data selector>          ; protected mode SS
DD    OFFSET <start>           ; protected mode EIP
DW    <code selector>          ; protected mode CS
```

Note that the data and code selectors used in the Application Initialization Block must reference valid entries in the GDT.

Procedure *CJ_PMSW* loads the GDT register, switches to protected mode and jumps to your startup code at the address specified in your Application Initialization Block.

Upon entry to your protected mode startup code, the following conditions exist.

> Interrupts are disabled.
> *CS:EIP* = start address
> *SS:ESP* = initial stack pointer
> *DS = ES = FS = GS = SS*
> *EAX* and *EBX* are undefined
> All other registers are unaltered from their state upon entry to *CJ_PMSW*.

Once your protected mode startup code gains control you can initialize the *IDTR* register with the address of the protected mode Interrupt Descriptor Table (IDT) using AMX procedure *CJ_PMIDTS*. The *IDTR* must be loaded before you launch AMX.

Note that, since the format of the Interrupt Descriptor Table changes from real mode to protected mode, you must leave interrupts disabled until the *IDTR* has been loaded in protected mode.

If your IDT is in RAM, you must initialize it yourself or launch AMX with access to the IDT (see the Selector Definition window in Chapter 4.2). You can then use procedure *cjksispwr* to install interrupt gates in the IDT for your interrupt devices. Procedure *cjksispwr* cannot be called before AMX is launched. If you are going to dynamically initialize your IDT in this fashion, be sure to launch AMX with interrupts disabled. Do your IDT initialization in a Restart Procedure which, by virtue of its position in your Restart Procedure List, is guaranteed to have the IDT initialization completed before interrupts are enabled.

D.4 Bootstrap Code Example

```
        TITLE    Sample Bootstrap Initialization
        NAME     BOOT
        .386P
;
;       ****************************************
;       *                                      *
;       *        AMX 386/ET Sample Real Mode   *
;       *        Bootstrap Initialization      *
;       *                                      *
;       ****************************************
;
;
; This code is an example of how to initialize the i386 processor
; to begin execution in protected mode.
;
;
        INCLUDE  CJ722K.DEF              ; AMX 386/ET Definitions
;
CODE  SEGMENT  BYTE PUBLIC USE16 'CODE'
        ASSUME CS:CODE, DS:DATA, ES:NOTHING, SS:NOTHING
;
        PUBLIC   BOOT                    ; Bootstrap entry point
;
        EXTRN    CJ_PMBLD:NEAR          ; build GDT
        EXTRN    CJ_PMSW:NEAR           ; load GDTR and switch to PM
;
; Bootstrap Procedure - Build protected mode tables and enter
;                       protected mode
;
BOOT  PROC      NEAR
        CLI                              ; disable interrupts
        MOV      AX,SEG DATA
        MOV      DS,AX                   ; set DS
        MOV      SS,AX
        LEA      SP,RMSP                 ; SS:SP = real mode stack
;
; Build GDT and enter protected mode
;
        MOV      AX,CS
        MOV      ES,AX
        LEA      SI,SDTABLE              ; ES:SI = A(Seg Desc Table)
        LEA      DI,GDTABLE              ; DS:DI = A(GDT)
        MOV      CX,GDTLEN               ; CX = size of GDT (bytes)
        CALL     CJ_PMBLD                ; build GDT
        LEA      SI,APIB                 ; ES:SI = A(App Init Block)
;                                        ; DS:DI = A(GDT)
        JMP      CJ_PMSW                 ; go to startup code in PM
;
BOOT  ENDP
;
        PAGE
```

```
        ; Segment Description Table
        ; Defines all of the segment descriptors that will be created
        ; in the Global Descriptor Table.
        ; Note that the first useable selector number is 10H.
        ; Selector 0 is not a valid selector.
        ; Selector 8 is reserved by AMX for use as a GDT data alias.
        ; This table assumes the Flat model.
        ;
SDTABLE  LABEL    BYTE
         DB       K_PMERCS            ; PM Code segment
         DB       0                   ; reserved
         DW       10H                 ; selector
         DD       0                   ; linear base address
         DD       0FFFFFFFFH          ; segment limit = 4GB
        ;
         DB       K_PMRWDS            ; PM Data segment
         DB       0                   ; reserved
         DW       18H                 ; selector
         DD       0                   ; linear base address
         DD       0FFFFFFFFH          ; segment limit = 4GB
        ;
         DB       0                   ; end of table
        ;
        ;
        ; Application Initialization Block
        ; Defines the initial stack pointer and start address for the code
        ; to be executed once the switch to protected mode occurs.
        ; This example assumes that the protected mode application has
        ; been separately linked using the Flat model.
        ; ROM is at offset 40000H and RAM is at offset 20000H.
        ;
APIB     LABEL    BYTE
         DD       21000H              ; stack pointer = ESP
         DW       18H                 ; data selector = SS
        ;                             ; DS,ES,FS,GS = SS
         DD       40000H              ; start address = EIP
         DW       10H                 ; code selector = CS
        ;
CODE     ENDS
        ;
DATA     SEGMENT  PARA PUBLIC USE16 'DATA'
        ;
GDTABLE  DB       8*4 DUP (?)         ; GDT RAM (selectors 0..18H)
GDTLEN   EQU      $-GDTABLE           ; length of GDT
        ;
         DB       32 DUP (?)          ; bootstrap stack
RMSP     LABEL    WORD
        ;
DATA     ENDS
        ;
CGROUP   GROUP    CODE                ; Code group
DGROUP   GROUP    DATA                ; Data group
        ;
         END      BOOT
```

D.5 Linking the Bootstrap System

Since the bootstrap code executes in real mode and the AMX system executes in protected mode, it is easiest to link and locate the final ROM in two stages in order to avoid mixed-mode confusion with the linker.

Use your linker and locator to produce an Intel absolute hex format file containing the real mode bootstrap code and an Intel 32-bit hex format file containing the protected mode code, both suitable for conversion to a ROM image.

The sample bootstrap code presented in Appendix D.4 is delivered to you as file *CJ722BSC.ASM*. Assume that you have used it to create and assemble a file *BOOT.ASM*. It is assumed that this bootstrap code is to be located in ROM at address *F000:0000* with data in RAM at *0000:0000*. Use your linker/locator to create a bootstrap ROM image file *BOOT.HEX*.

Then link and locate your protected mode AMX application into ROM image file *APPLN.HEX*. Note that the bootstrap code example assumes that your application uses the flat model with code in ROM at offset *40000H* and data in RAM at offset *20000H*. The protected mode application is entered at its start address of *10:40000* with a stack set to be *18:21000*.

Hex files *BOOT.HEX* and *APPLN.HEX* can then be burned in ROM.

Appendix E. Board Support Porting Issues

E.1 Porting the AMX 386/ET Sample Program

AMX 386/ET is delivered to you with a Sample Program ready for use on an Intel386™, Intel486™ or Pentium™ hardware platform with a PC/AT-like architecture. For example, the Sample Program can be used on a conventional PC treated as an embedded target.

Unfortunately, not all PCs are alike so you may have to make some modifications to the Sample Program for it to work with your target hardware and your debugger. It is the purpose of this appendix to help you determine what, if anything, needs to be modified.

On the PC/AT, the Intel 8253 timer interrupts on IRQ0 of the master 8259 interrupt controller which uses a block of eight interrupt descriptors beginning at entry 8 in the Interrupt Descriptor Table. However, the use of that particular block of eight descriptors for device interrupts conflicts with the processor's use of the vectors for the detection of processor dependent faults. Consequently, on some boards such as the Intel386EX Evaluation Board, a ROM monitor relocates the block of eight descriptors from entry 8 to some other entry (often entry 32) in the Interrupt Descriptor Table. A similar relocation may also be done by a debug monitor (such as Paradigm's *PDREMOTE/ROM* Target Monitor) which permits a PC/AT to be used as a target embedded system.

The Sample Program uses one Intel 8250 serial port for access to a simple display terminal. The serial device driver provided with the Sample Program is configured to support two 8250 UARTs located at the standard PC/AT device addresses of *0x3F8* for COM1 and *0x2F8* for COM2 . The Sample Program then assumes that only the second port (COM2) is available for display since the first port (COM1) is often used by a debug monitor for communication with a source level debugger operating on a connected host PC or work station. Unfortunately, some boards such as the Intel386EX Evaluation Board, expect the debug monitor to use COM2 leaving COM1 available for use by the application.

If you installed AMX into directory *C:\KADAK* then the files which may require modification will be found in directory *C:\KADAK\AMX722\TOOLXX\SAMPLE* where *XX* is the KADAK mnemonic for the supported software development toolset combination which you are using. If you are using the Intel386EX Evaluation Board or a PC/AT, use the files in board support subdirectory *AT386*. The files of interest are the clock driver *CH8253T.C* and the serial port driver *CH8250S.C*.

Verify that the clock driver `CH8253T.C` uses the correct interrupt descriptor number for the 8253 timer in your hardware configuration. Look for and, if necessary, modify the following definition.

```
#define VVINTV 0x40
```

Verify that the serial driver `CH8250S.C` assigns the correct 8250 serial port for use by your debugger. Look for and, if necessary, modify the following definition.

```
#define VVDEBUG 1
```

The serial driver `CH8250S.C` supports the serial ports in the Intel386EX chip or in a conventional PC/AT hardware configuration. Two symbols (`VV386EX` and `VVPCAT`) are used to select the intended target hardware. One, and only one, of these symbols must be defined.

To use serial driver `CH8250S.C` with the Intel386EX Evaluation Board, omit any definition of `VVPCAT` and include the following definition.

```
#define VV386EX
```

To use serial driver `CH8250S.C` on a PC/AT, omit any definition of `VV386EX` and include the following definition. The number of PC/AT COM ports (1 to 4) to be supported is provided in the definition.

```
#define VVPCAT 2
```

If you use any of these toolsets	on this hardware platform	VVINTV must be	VVDEBUG must be	VVboard must be
with the Paradigm Debugger:				
PD	PC/AT	0x40	1	VVPCAT 2
PD	Intel386EX Eval Board	0x40	2	VV386EX

KADAK

E.2 Spurious Interrupts and the 8259 PIC

If your target hardware includes one or more Intel 8259 interrupt controllers (or functional equivalents), you must be certain to account for the manner in which the controllers cope with spurious interrupt requests. This note therefore applies to the Intel386EX™ processor with its two internal controllers and to any PC/AT-like target hardware configuration.

When the 8259 interrupt controller detects an interrupt request which is removed before it can be acknowledged, it declares the interrupt to be spurious. In response to a spurious interrupt, the controller generates an IRQ7 interrupt but does NOT set the corresponding in-service bit in its In-Service Register (ISR).

Because of this feature, it is essential that you provide an Interrupt Service Procedure (ISP) for IRQ7 of every 8259 interrupt controller which exists in your hardware configuration. If the master controller's IRQ7 interrupt is not used, your ISP can simply issue an *IRETD* instruction to ignore the interrupt. If a slave controller's IRQ7 interrupt is not used, your ISP must clear the master controller's interrupt request from the slave and then issue an *IRETD* instruction to ignore the interrupt.

AMX 386/ET includes default spurious interrupt handlers for both master and slave 8259 interrupt controllers. To include these handlers in your AMX system, use the AMX Configuration Manager to add the following statements to your AMX 386/ET Target Parameter File.

```
...M8259    M8259ISP
...S8259    S8259ISP,MPORT,SPORT,EOI-CMD
```

M8259ISP is the name to be assigned to your master 8259 ISP.
S8259ISP is the name to be assigned to your slave 8259 ISP.
MPORT is the master 8259 device address.
SPORT is the slave 8259 device address.
EOI-CMD is the 8259 command byte to be issued to the master 8259
     to clear a spurious interrupt occurring on the slave 8259.

If IRQ7 is used on the master 8259 interrupt controller, omit the *...M8259* directive. If you have no slave 8259 interrupt controller, omit the *...S8259* directive. If you have more than one slave controller, you will need a separate *...S8259* directive for each slave. If IRQ7 is used on a slave 8259 interrupt controller, omit the *...S8259* directive for that slave.

When your AMX system is launched, you must update the Interrupt Descriptor Table so that the entries for the master and slave 8259 IRQ7 interrupts reference the default ISPs in your Target Configuration Module.

Example: Application Ignores IRQ7

The following is an example of an AMX application operating on a PC/AT or compatible hardware platform. It is assumed that the application does not use IRQ7 on either the master or slave 8259 interrupt controller.

To include the default spurious interrupt handlers for both the master and slave controllers, insert the following directives into your AMX 386/ET Target Parameter File.

```
...M8259    ATMaster
...S8259    ATSlave,20H,0A0H,20H
```

The resulting Target Configuration Module will include two Interrupt Service Procedures. The ISP at *ATMaster* will ignore spurious interrupts from the master 8259. The ISP at *ATSlave* will ignore spurious interrupts from the slave 8259.

Create the following Restart Procedure and add *rr8259* at or near the beginning of your list of Restart Procedures in your AMX Configuration Module. The Restart Procedure initializes the entries in the Interrupt Descriptor Table (IDT) for IRQ7 of the master and slave 8259 interrupt controllers.

```
#include "CJZZZ.H"

void CJ_CCPP ATMaster(void);        /* Master 8259 ISP             */
void CJ_CCPP ATSlave(void);         /* Slave 8259 ISP              */

#define M8259BASE 0x08              /* Master 8259 IRQ0 base       */
#define S8259BASE 0x70              /* Slave 8259 IRQ0 base        */

void CJ_CCPP rr8259(void)
{
                                    /* Install Master 8259 IRQ7 ISP  */
   cjksispwr(M8259BASE + 7, (CJ_ISPPROC)ATMaster);

                                    /* Install Slave 8259 IRQ7 ISP   */
   cjksispwr(S8259BASE + 7, (CJ_ISPPROC)ATSlave);
   }
```

Note that the Restart Procedure in this example unconditionally installs interrupt gates to the ISPs into the IDT as is appropriate for an AMX application launched for permanent execution. If your AMX application is launched for temporary execution, you should save the initial values of each IDT entry using *cjksidtrd*. Then create an Exit Procedure which uses *cjksidtwr* to restore the IDT entries to their initial values when your AMX application shuts down.

Example: Application Uses IRQ7

If your application uses IRQ7 on any 8259 interrupt controller, your Interrupt Service Procedure (ISP) for that interrupt MUST account for the possibility that the interrupt request was spurious. To do so, your ISP must read the 8259 In-Service Register (ISR) and examine ISR bit 7. If the bit is 0, the IRQ7 interrupt request is spurious and must be ignored as described previously. If the bit is 1, your ISP must service the real IRQ7 interrupt request.

To read the 8259 In-Service Register you must first select the ISR and then read it. For example, if the 8259 device address is *0x20*, write *0x0B* to output port *0x20* and then read input port *0x20*. This write/read operation MUST be done with interrupts disabled.

If your ISP enables interrupts, then it must be prepared to accept a spurious interrupt on IRQ7 while it is servicing a real IRQ7 interrupt. That is, the ISP must be recursive. Since bit 7 in the In-Service Register remains set while the real IRQ7 is being serviced, the bit can no longer be used to detect a spurious interrupt. The following example of a conforming AMX Interrupt Handler for IRQ7 on the master 8259 illustrates this requirement.

```
#include "CJZZZ.H"
#define M8259 0x20                    /* Master 8259 device address    */
#define M8259BASE 0x08                /* Master 8259 IRQ0 base         */


void CJ_CCPP IRQ7root(void);          /* ISP root for IRQ7 ISP         */
static int in_service;                /* Private boolean               */


void CJ_CCPP IRQ7rr(void)             /* IRQ7 Restart Procedure        */
{
   in_service = 0;                    /* IRQ7 is not in service        */
                                      /* Install IRQ7 ISP root         */
   cjksispwr(M8259BASE + 7, (CJ_ISPPROC)IRQ7root);
   }


void CJ_CCPP IRQ7isp(void)            /* AMX Interrupt Handler for IRQ7 */
{
   if (in_service)
          return;                     /* In service; must be spurious  */

   cjcfoutp8(M8259, 0x0B);
   if ( (cjcfinp8(M8259) & 0x80) == 0)
          return;                     /* Not a real IRQ7               */

   in_service = 1;                    /* Servicing a real IRQ7         */
   cjcfei();                          /* Enable interrupts             */
   :
   : Service the device and remove the level 7 interrupt request
   :
   cjcfdi();                          /* Disable interrupts            */
   in_service = 0;                    /* Not in service                */
   cjcfoutp8(M8259, 0x20);            /* Non-specific EOI              */
   }
```

This page left blank intentionally.