AMX®

AMX™ User's Guide

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

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

TECHNICAL SUPPORT

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

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

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

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

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

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

AMX USER'S GUIDE

Table of Contents

AMX USER'S GUIDE

Table of Contents (Cont'd.)

AMX USER'S GUIDE
Table of Contents (Cont'd.)

AMX USER'S GUIDE
Table of Figures

1. AMX Overview

1.1 Introduction

The AMX™ Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. AMX allows the system designer to concentrate on the application without becoming overly concerned about the multitasking aspects of the solution.

AMX is based on concepts proven over the past thirty years in minicomputer and microprocessor applications in process control environments. AMX simplifies real-time software implementation by providing the system designer with a well-defined set of rules.

AMX gives the system designer complete flexibility and control over the microcomputer configuration employed. A real-time clock must be provided in the configuration if AMX timing facilities are to be employed.

AMX provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your AMX system. The AMX managers are all optional.

The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement an AMX-based real-time operating system. It is assumed that you are familiar with the architecture of the processor on which you will be using AMX. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation.

AMX is available in C source format to ensure that regardless of your development environment, your ability to use and support AMX is uninhibited. The source program also includes a small portion programmed in the assembly language of the target processor.

The C programming language, commonly used in real-time systems, is used throughout this manual to illustrate the features of AMX.

Section Summary

This manual is divided into three sections. Each section is divided into chapters.

Section 1 of this manual describes the AMX Multitasking System and how it is used. Separate chapters are provided for each of the AMX managers.

Section 2 describes the AMX System Configuration Builder and the manner in which it is used to create your AMX System Configuration Module and Target Configuration Module.

Section 3 is the application programming guide. It provides detailed descriptions of all of the AMX service procedures which are available in the AMX Library.

AMX Guides

Guidelines for the installation and proper use of AMX when coding in C are provided in the separate **Getting Starting** guide. The Getting Started guide includes a description of the AMX Sample Program provided with AMX as well as useful debugging tips.

This manual describes the use of AMX for all target processors. Target specific requirements or programming considerations are provided in a separate **AMX Target Guide**.

This manual describes the use of AMX in a toolset independent fashion. References to specific assemblers, compilers, librarians, linkers, locators and debuggers are purposely omitted. The **AMX Tool Guide** provides guidance for the proper use of AMX with each toolset with which AMX has been tested.

The **AMX Timing Guide** discusses general timing issues related to the use of AMX. Timing metrics generated for specific boards and software development toolsets are also provided. These timing figures can be used as guidelines to expected AMX performance, but are not to be construed as product specifications. The AMX Timing Guide is packaged with AMX as an on-line document. The manual is not provided in printed form.

1.2 Glossary

Binary and Bounded Semaphores	A bounded semaphore is a counting semaphore whose signal count can range from 0 to an upper limit less than or equal to 16383. If the upper limit is 1, the semaphore is a binary semaphore.
Buffer Pool	A collection of data buffers whose use is controlled by the AMX Buffer Manager.
Buffer Pool Id	The handle assigned to a buffer pool by AMX for use as a unique buffer pool identifier.
Circular List	An application data structure used to maintain a list of 1, 2 or 4 byte elements with the ability to add and remove elements at both the top (head) and bottom (tail) of the list.
Clock Handler	The name given to the AMX procedure which is called by the ISP root which services the hardware clock interrupt.
Clock Tick	The interrupt generated by a hardware clock.
Conforming ISP	An Interrupt Service Procedure consisting of an ISP root which calls an Interrupt Handler which has the right to make calls to a subset of the AMX service procedures.
Counting Semaphore	A particular type of AMX semaphore used for event signalling or for controlling access by tasks to multiple resources. A counting semaphore is limited to a maximum count of 16383. Binary semaphores and bounded semaphores are counting semaphores with a fixed upper limit.
Envelope	The private data storage element used by AMX to pass a message to a mailbox or message exchange.
Error Code	A series of signed integers used by AMX to indicate error or warning conditions detected by AMX service procedures.
Event Group	A set of 16 or 32 events whose access and signalling is controlled by the AMX Event Manager.
Event Group Id	The handle assigned to an event group by AMX for use as a unique event group identifier.
Exit Procedure	An AMX or application procedure executed by AMX during the exit phase when an AMX system is shut down.
Fatal Error	A condition detected by AMX which is considered so abnormal that to proceed might risk catastrophic consequences. Examples include, but are not limited to, insufficient memory in the AMX Data Segment or division by zero in an ISP.
FIFO	First in, first out. Usually used to refer to the ordering of elements in a queue, circular list or linked list.

Group Id	See Event Group Id.
Handle	An identifier assigned by AMX for use by your application to reference a private AMX system data item.
Interrupt Handler	An application procedure called from an ISP root to service an interrupting device.
Interrupt Service Procedure (ISP)	An AMX or application procedure which is executed in response to an external device interrupt request.
ISP	See Interrupt Service Procedure
ISP root	The ISP code fragment (produced by the AMX Configuration Generator) which informs AMX that an interrupt has occurred and calls an application Interrupt Handler.
Kernel Task	The private AMX task which is responsible for all timing control and event sequencing in an AMX system.
Linked List	An application data structure used to maintain a doubly-linked list of arbitrary application data elements with the ability to add and remove elements at head, tail or specified positions in the list.
List Element	An 8-bit, 16-bit or 32-bit value which can be added to or removed from a circular list.
Mailbox	An AMX data structure consisting of a single message queue. Mailboxes allow the interchange of messages between two or more cooperating components (tasks, ISPs, etc.) of an AMX system.
Mailbox Id	The handle assigned to a mailbox by AMX for use as a unique mailbox identifier.
Memory Block	A portion of a memory pool that has been allocated for use by one or more tasks.
Memory Pool	A collection of memory sections whose use is controlled by the AMX Memory Manager.
Memory Pool Id	The handle assigned to a memory pool by AMX for use as a unique memory pool identifier.
Memory Section	A contiguous region of memory assigned to the AMX Memory Manager for allocation to application tasks.
Message	Application information passed by AMX in an envelope to a mailbox or message exchange.
Message Exchange	An AMX data structure that consists of four message queues, each for messages of a different priority. The message exchanges allow the interchange and prioritization of messages between two or more cooperating components (tasks, ISPs, etc.) of an AMX system.

Message Exchange Id	The handle assigned to a message exchange by AMX for use as a unique message exchange identifier.
Message Exchange Task	An application task to which a private message exchange has been attached such that the task automatically receives messages from the exchange.
Message Queue	An AMX data structure used to manage messages sent to a message exchange. A separate message queue is provided for each of the four message priorities which a message exchange can support.
Message Priority	Identifies which of a message exchange's four message queues is to receive the AMX message.
Nonconforming ISP	An Interrupt Service Procedure which bypasses AMX (has no ISP root) and hence cannot use any AMX service procedures.
RAM	Alterable memory used for data storage and stacks.
Resource Semaphore (Basic and Priority Inheritance)	A particular type of AMX semaphore used to provide access to an entity such as a math coprocessor, disk file or non-reentrant library whose ownership is to be controlled by the AMX Semaphore Manager. A resource semaphore which supports priority inheritance can be used to avoid task priority inversions.
Restart Procedure	An AMX or application procedure executed by AMX during the initialization phase when an AMX system is started.
ROM	Read only memory of all types including PROMs, EPROMs and EAROMs.
Segment	An area of memory in which AMX code or data is stored. Segments are sometimes called sections or regions according to the nomenclature adopted for a particular processor.
Semaphore	An AMX data structure which can be used by the AMX Semaphore Manager to provide an event signalling mechanism or mutually exclusive access by tasks to specific user facilities.
Semaphore Id	The handle assigned to a semaphore by AMX for use as a unique semaphore identifier.
Slice Interval	The interval of time allocated to a task which is time sliced.
Slot	One of *n* locations used to store list elements in a circular list.
System Configuration Module	A software module, produced by the AMX System Configuration Builder, which defines the characteristics of a particular AMX application.

System Tick	A multiple of the hardware clock tick from which the fundamental AMX unit of time is derived. All time intervals in an AMX system are measured in multiples of the system tick.
Tag	A 4-character name that can be assigned to any AMX system data structure when it is created. A tag can be used to find the identifier of a task, timer, semaphore, event group, mailbox, message exchange, memory pool or buffer pool with a particular name.
Target Configuration Module	A software module, produced by the AMX Configuration Generator, which defines the characteristics of your target hardware as used in a particular AMX application.
Task	An application procedure which is executed by AMX in a way which makes it look as though all such procedures are executing at once.
Task Control Block (TCB)	A private data structure maintained by AMX for each task in the system.
Task Id	The handle assigned to a task by AMX for use as a unique task identifier.
Task Priority	The priority at which a task executes. Tasks which have the same task priority are actually ordered in priority according to the order in which the tasks were created, barring changes in priority caused by time slicing or the use of priority inheritance resources.
TCB	See Task Control Block
Time Slice	The process by which AMX allows tasks having the same priority to share the use of the processor in a round robin fashion.
Timer	A facility provided by AMX to permit precise interval measurement in AMX applications.
Timer Id	The handle assigned to a timer by AMX for use as a unique timer Identifier.
Timer Procedure	An application procedure which is executed by the AMX Kernel Task whenever the corresponding timer interval expires.

1.3 AMX Nomenclature

The following nomenclature standards have been adopted throughout the AMX User's Guide.

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

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

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

AMX symbol names and reserved words are identified as follows:

cjkkpppp	AMX C procedure name *pppp* for service of class *kk*
cjxtttt	AMX structure name of type *tttt*
xttttyyy	Member *yyy* of an AMX structure of type *tttt*
CJ_ID	AMX object identifier (handle)
CJ_ERRST	Completion status returned by AMX service procedures
CJ_CCPP	Procedures use C parameter passing conventions
CJ_ssssss	Reserved symbols defined in AMX header files
CJ_ERXXXX	AMX Error Code *XXXX*
CJ_WRXXXX	AMX Warning Code *XXXX*
CJ_FEXXXX	AMX Fatal Exit Code *XXXX*
CJZZZFFF.XXX	AMX filenames
CJZZZ.H	Generic AMX include file

The *ZZZ* in each AMX filename is the 3-digit AMX part number used by KADAK to identify the AMX target processor. For example, file *CJZZZSD.H* is an AMX header file for any version of AMX. File *CJ999SD.H* is the same AMX header file for the version of AMX which operates on the processor identified by KADAK part number *999*.

File *CJZZZ.H* is a generic include file which includes the subset of target specific AMX header files needed for compilation of your application C code. By including file *CJZZZ.H* in your source modules, your AMX application becomes readily portable to other target processors.

The generic include file *CJZZZ.H* is a copy of the corresponding part numbered AMX file. For example, if you are developing for the processor identified by KADAK part number *999*, the file *CJZZZ.H* is a copy file *CJ999.H*.

Throughout this manual examples are provided in C. In general, code examples are presented in lower case. File names are shown in upper case. C code assumes that an *int* is 32 bits on 32 bit processors or 16 bits on 16 bit processors as is common for most C compilers.

This page left blank intentionally.

2. General AMX Operation

2.1 Introduction to Multitasking

A real-time system is characterized by the need to respond rapidly to events occurring asynchronously in time. A multitasking system is one in which a number of activities or processes must be performed simultaneously without interference with each other. A system in which several activities must operate simultaneously with time-critical precision is called a real-time multitasking system.

The AMX Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. Each task solves a particular problem and provides a specific functional capability within the system.

As we all know, the microprocessor can only do one thing at a time. Fortunately, it does all things very quickly. However, to get the effect that all activities are occurring simultaneously, it is necessary to rapidly switch back and forth from one process to another in a well controlled fashion. It is AMX which organizes and controls the use of the microprocessor to achieve this apparent concurrent execution of tasks.

Each task solves a particular problem or provides a specific functional capability within the system. Each task executes independent of other tasks. Facilities are provided, however, to permit tasks to co-operate to achieve a common goal. This process in which more than one task is allowed to share the use of a single processor is called multitasking. The software program which makes it possible is AMX.

What a task does is completely application dependent. In fact, the most difficult aspect of system design is to logically break the problem into a set of tasks which, if implemented, will achieve the desired goal. The following example is presented to illustrate one way in which a simple real-time alarm logging system can be implemented.

Example

Assume that it is necessary to periodically scan a set of digital alarm inputs. When any alarm is detected, a message is to be logged on a printer. The message is to include a description of the alarm and the time of day at which it occurred.

Careful examination of this problem indicates that in fact it is three problems. First, a set of digital inputs must be scanned for the detection of alarms. Second, a message must be printed. Finally, the time of day must be maintained for inclusion in the message.

Three problems usually result in three tasks. Our example is no exception. A time of day task is required to maintain the date and time within the system. The task must be executed once each second if time is to be maintained to the nearest second. We will ignore the requirement to somehow set the proper time and date. (Isn't that another task?)

Alarm scanning will likely be hardware dependent. We will simplify matters by assuming that a scanning task must examine all digital inputs every 100 ms. The task must be capable of detecting alarm changes in the digital inputs. When an alarm is detected, the scanning task must initiate the logging of a message.

However, the scanning task is not allowed the luxury of waiting until the message is printed. It must continue scanning for additional alarms at the same 100 ms rate. The scanning task must, therefore, send to the message task parameters identifying the alarm which has occurred and the time at which it was detected.

The message task is very simple to describe. It receives parameters identifying the time at which a particular alarm occurred. The task must output to a printer a message describing the alarm.

The foregoing example, although very simple, illustrates many of the features of a real-time multitasking system. At first, the implementation of the required set of tasks may be difficult to envision. Two of the tasks must operate periodically at different intervals. The third task, printing, is such a slow process that the other two tasks must be allowed to execute with higher priority. Moreover, provision must be made to allow the alarm parameters to continue piling up on the message task when alarms are occurring at a rate faster than can be printed. (What would we do if we had to print high priority alarms first and delay the printing of lower priority alarms?)

The implementation of the proposed solution is greatly simplified by AMX. AMX will supervise the execution of the three tasks in the defined priority order. Timing facilities are provided by AMX with a resolution governed by the hardware clock. AMX supports message passing between tasks. AMX allows these messages to pile up at the destination message exchange at any of four priority levels because automatic message queuing with priority sorting is an inherent feature of AMX.

This introduction to multitasking is not exhaustive. The intent is to remove some of the mystery from the multitasking concept. The above example is intended to inspire confidence in your ability to understand AMX and use it to solve real-time problems.

2.2 AMX Operation

AMX Startup

Each AMX-based system consists of the AMX executive program and a set of application tasks and interrupt service procedures. This collection of programs resident in the memory of the microprocessor configuration represents the entire operating system.

The manner in which the operating system begins execution is application dependent. In ROM-based systems, automatic hardware vectoring to the program start address is often implemented. In RAM-based systems, the program is first loaded from some storage medium (ROM, hard disk, diskette, etc.) or downloaded from one processor to another. Once the program is loaded, it is started at its start address by the loader.

Figure 2.2-1 illustrates the general operation of an AMX system. Execution begins in the user domain providing the opportunity for hardware specific and application dependent setup prior to the initialization of the AMX system. For example, hardware interfaces may require custom configuring. In some systems, it might be desirable to perform a memory integrity check before system startup is permitted.

Once all custom initialization has been performed, the program calls the AMX entry procedure *cjkslaunch*. Operating characteristics are defined in an AMX System Configuration Module and an AMX Target Configuration Module. It is possible to predefine specific tasks and timers which will be automatically created by AMX during its initialization phase. AMX initializes itself and places all application tasks and timers into an idle state.

Once AMX has initialized all of its internal variables and structures, it executes a sequence of user provided Restart Procedures. These procedures can invoke AMX services to start tasks and initialize interval timers.

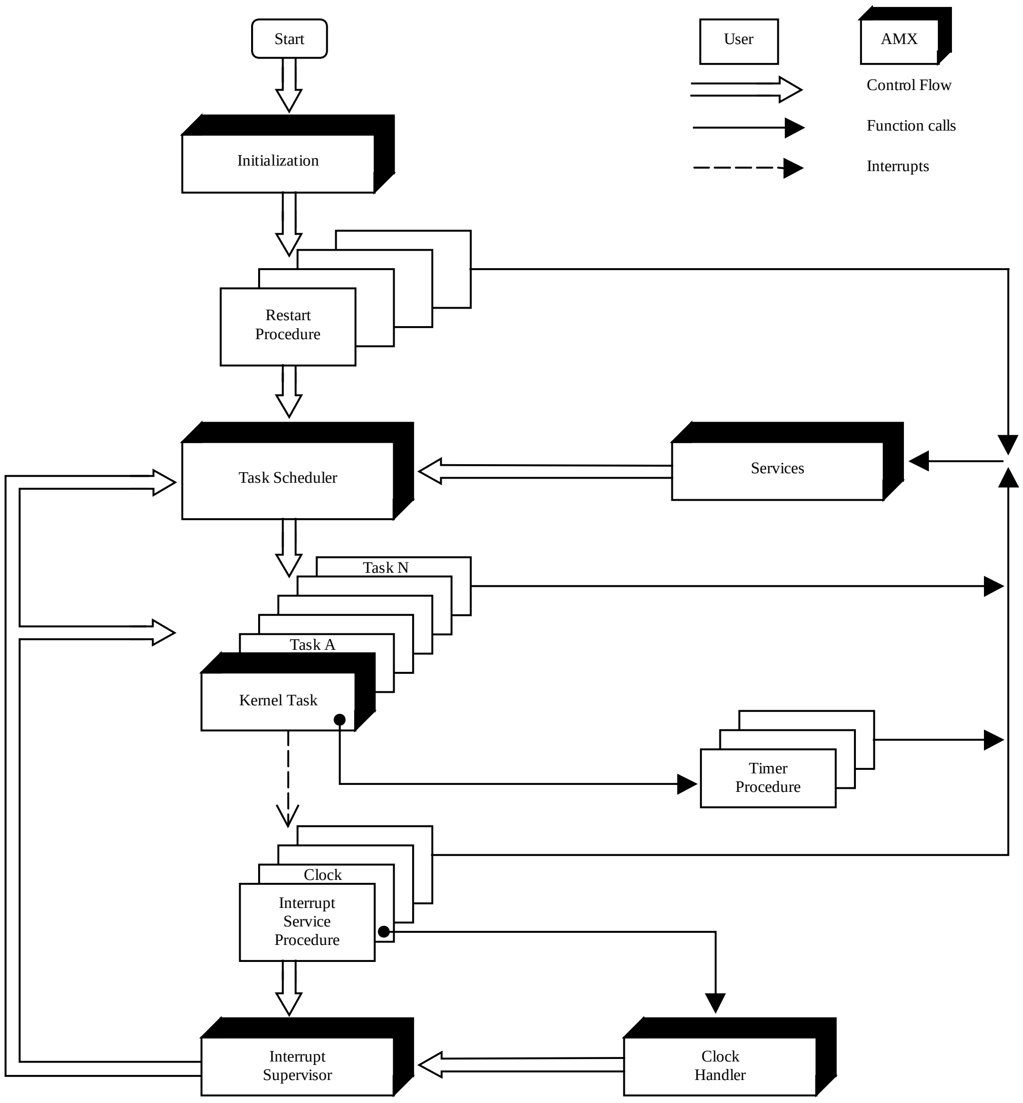


Figure 2.2-1 AMX General Operation

The Task Scheduler

Following system initialization, AMX proceeds to its Task Scheduler. The Task Scheduler searches its list of available tasks to determine the highest priority task capable of execution. Task execution priorities are determined by the system designer. If no task is ready to begin execution, AMX sits with interrupts enabled, waiting for some external event to generate an interrupt.

AMX begins task execution at the task's defined start address. The task executes as though it were the only program in the system. Services offered by AMX can be invoked by the task by procedure calls as indicated in Figure 2.2-1.

Once a task begins execution, it appears to operate without interruption. The interrupts that are periodically taking place are completely hidden from the task by the AMX Interrupt Supervisor and Task Scheduler. The task, once executing, inhibits the performance of all tasks of priority lower than its own. The task continues to execute until it decides to relinquish control, even if only temporarily, by calls to AMX.

The task ends by returning to the AMX Task Scheduler which again finds the next highest priority task ready to execute and gives it control of the processor. A task, once executing, is free to call any of the AMX task services. For instance, a task can send a message to a mailbox or message exchange, wait for an event or wait for a timed interval. If the task wishes to wait for an event, the AMX service procedure will suspend the task and request the AMX Task Scheduler to force execution of the next highest priority task ready for execution.

AMX acts as the context switcher supervising the orderly execution of application tasks. AMX employs a preemptive, priority-driven scheduling algorithm which ensures that the highest priority task which is ready to do useful work always has control of the processor.

AMX will switch tasks if it receives a request from the executing task to perform an operation which, of necessity, invokes a task of higher priority. For instance, the executing task may request AMX to start a higher priority task.

The Interrupt Supervisor

Tasks execute with the processor interrupt facility enabled to permit service of external devices. When an external interrupt occurs, the task is interrupted in the manner dictated by the processor. The processor automatically saves the return address and some subset of the processor state (registers, flags, etc.) and branches to an Interrupt Service Procedure (ISP). The exact vectoring method is determined by the hardware configuration employed in the system.

Two types of ISPs exist: nonconforming ISPs and conforming ISPs.

A **nonconforming ISP** must quickly service the device to remove the interrupting condition. The ISP must preserve all registers which it uses. The nonconforming ISP cannot make calls to any AMX service procedures.

A **conforming ISP** can make use of a subset of the AMX service procedures. A conforming ISP consists of an ISP root and an Interrupt Handler. The processor vectors to the ISP root which informs the AMX Interrupt Supervisor that the interrupt has occurred. The Interrupt Supervisor preserves the state of the interrupted task and, if necessary, switches to an interrupt stack. The Interrupt Supervisor then calls the associated Interrupt Handler.

The Interrupt Handler must quickly service the device to remove the interrupting condition. The handler is free to make procedure calls to a subset of the AMX service facilities. When device service is completed, the AMX Interrupt Supervisor dismisses the interrupt.

The AMX Interrupt Supervisor monitors calls made by the Interrupt Handler to AMX service procedures. If no such calls have been made, AMX automatically restores the state of the interrupted task and returns directly to the interrupted task at its point of interruption.

The Interrupt Handler may have requested AMX to initiate or resume execution of some task of higher priority than the interrupted task. If so, the AMX Interrupt Supervisor suspends the interrupted task and marks it as ready to resume execution at the earliest opportunity. The AMX Task Scheduler is then invoked to determine the highest priority task capable of execution.

The AMX Interrupt Supervisor supports nested interrupts on processors which provide this capability. If interrupts nest, the Interrupt Supervisor defers its task switching checks until all of the concurrent interrupts have been serviced.

Note

The conforming ISP root is created by the AMX Configuration Generator in your AMX Target Configuration Module.

Timing Facilities

The AMX Timer Manager provides a Clock Handler and a Kernel Task to provide complete timing control for your real-time application. The AMX Clock Handler is independent of any particular hardware configuration. If AMX timing facilities are to be utilized, a real-time clock must be included in the configuration.

The hardware clock interrupt must be serviced by a conforming ISP consisting of an ISP root and a clock Interrupt Handler. The ISP root calls your clock Interrupt Handler to dismiss the clock interrupt. It then calls the AMX Clock Handler to derive an AMX system tick.

The AMX Clock Handler triggers the AMX Kernel Task if required. The Kernel Task is triggered at the user defined system tick interval if, and only if, there is any outstanding timing activity required in the system. In this case, the interrupted task is suspended and the AMX Kernel Task begins execution.

The AMX Kernel Task executes as the highest priority task in the system. The AMX Kernel Task monitors all tasks which are in a timed wait state. If a task's timer expires, the AMX Kernel Task primes the task to resume execution with a timeout indication.

The AMX Kernel Task also services all expiring application interval timers. Whenever an application interval timer expires, the corresponding application Timer Procedure is executed. This procedure can invoke a subset of the AMX services to send messages, signal events or wake tasks. If the timer is defined to be periodic, the AMX Kernel Task automatically restarts it with the predefined period.

Message Queuing

One of the more powerful features of AMX is its ability to queue messages for tasks. The queuing facility permits messages to pile up in a controlled fashion, freeing the ISP, Timer Procedure or task which is sending the message to continue with its appointed function. If a task sends a message, it can suspend itself until the message has been received and processed by some other task.

The AMX message queuing facility is further enhanced by allowing the messages to queue at any of four priority levels. A task can therefore receive messages from a variety of callers with the messages already sorted in order of priority by AMX.

The system designer describes the exact message queuing requirements in a mailbox or message exchange definition. For each message exchange, you can specify which, if any, of the four message queuing priority levels are to be supported. For each mailbox, only one message queue is provided. You also specify the required message nesting depth for each of these message queues.

AMX maintains a free list of message envelopes. These envelopes are used by AMX to transmit messages to mailboxes and message exchanges. The caller's message parameters are moved into a free envelope which is then added to the message queue of the destination mailbox or message exchange.

The AMX message queuing facility is especially useful in event logging applications. In such applications, messages are transmitted to a printing exchange by any task wishing to log an event. The printing task waits on the printing exchange and processes the requests as they become available. High priority messages can easily be forced to precede low priority messages using the AMX message priority feature. Finally, any task wishing to wait until its particular message has been logged can do so by informing AMX of its intent at the time the message is sent.

AMX Shutdown

An AMX system can be shut down in the same orderly fashion in which it was started. A task initiates a shutdown with a call to the AMX exit procedure *cjksleave*.

It is the caller's responsibility to assure that the AMX system is in a reasonable state to permit a shutdown. For instance, all device I/O operations should be stopped and all timing activity should be curtailed.

The AMX shutdown procedure executes a series of user Exit Procedures which can restore the original environment which existed prior to starting the AMX system. Device interfaces can be reprogrammed and interrupt vectors restored if necessary.

Once all of the Exit Procedures have been executed, AMX returns to the point at which AMX was originally launched.

2.3 AMX Managers

AMX provides a set of managers to simplify event synchronization, resource manipulation and memory allocation. Not all applications will make use of all of the managers. The system designer can decide which of the AMX managers is best suited for a particular application.

The **Time/Date Manager** provides Y2K compliant time of day calendar support if required. The AMX calendar clock includes second, minute, hour, day, month, year and day of the week. AMX services are provided to set and read the calendar clock. A formatting procedure is also provided to translate the calendar time and date from the internal format in which it is maintained by AMX into an ASCII string in several of the most popular formats.

An application procedure can be tied to the calendar clock and called at one second intervals to permit simple time of day event scheduling.

The **Semaphore Manager** provides two types of semaphores each with priority queuing and timeout: resource semaphores and counting semaphores.

A resource semaphore can be used to provide controlled access to your resources. It uses a binary semaphore to limit access to each resource to one task at a time. Ownership and release of a resource is governed by calls to the Semaphore Manager. A resource semaphore offers the unique characteristic of identifying each resource owner. Only the task which owns a resource is permitted to release it.

General purpose counting semaphores can be created for mutual exclusion and resource management. Tasks must request the Semaphore Manager for access to the resource controlled by such a semaphore. The task can specify the priority of its request to use the semaphore. If the semaphore is not free, the task will be forced to wait for its availability. The task will be placed on the semaphore wait queue at the priority specified by the task. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait.

A task, ISP or Timer Procedure can signal the semaphore with a call to the Semaphore Manager. The Semaphore Manager grants access to the semaphore to the task, if any, waiting at the top of the semaphore's wait queue.

The **Event Manager** provides a convenient method for synchronizing one or more tasks to events detected in Interrupt Service Procedures, Timer Procedures and other tasks. A task requests the Event Manager to suspend its operation until any one of a particular set of events occurs. Alternatively, the task can request to wait until all of a set of event conditions are met. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait. More than one task can be waiting for the same event or set of events.

When a task, ISP or Timer Procedure detects an event, it signals the event with a call to the Event Manager. The Event Manager checks to see if the event has resulted in an event combination for which one or more tasks are waiting. If so, the tasks which were waiting are allowed to resume execution.

The **Mailbox Manager** provides a general purpose message queuing facility. Tasks, ISPs or Timer Procedures can send messages to a mailbox. The messages are ordered in the mailbox according to their order of arrival. When a task requests a message from a mailbox, it is given the first message in the mailbox. If no message is available, the task will be forced to wait. The task will be placed on the mailbox wait queue at the wait priority specified by the task. The task can specify a timeout interval limiting the time the task is prepared to wait. This interval can be from no wait to an indefinite wait. When a message subsequently arrives, it is immediately given to the task waiting at the top of the mailbox's wait queue.

The **Message Exchange Manager** provides a general purpose prioritized message queuing facility. Tasks, ISPs or Timer Procedures can send messages to a message exchange to be queued at any of four priority levels. When a task requests a message from a message exchange, it is given the highest priority message available. If no message is available, the task will be forced to wait. The task will be placed on the message exchange wait queue at the wait priority specified by the task. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait. This interval can be from no wait to an indefinite wait. When a message subsequently arrives, it is immediately given to the task waiting at the top of the message exchange's wait queue.

The **Buffer Manager** provides fast, efficient access to multiple pools of buffers, each buffer representing a fixed size block of memory. This form of memory management meets the requirements of most typical applications and is best suited for real-time use in which memory block availability must be predictable and in which the penalties for memory fragmentation cannot be tolerated.

Application modules can request the Buffer Manager to get a buffer from a pool. If no buffer is available, a task can wait for a buffer to become available. Optionally, the task can specify a timeout interval limiting the time the task is prepared to wait. This interval can be from no wait to an indefinite wait. When a buffer subsequently becomes available, it is immediately given to the task waiting at the top of the buffer pool's wait queue.

When released, the buffer is automatically returned by the Buffer Manager to the pool to which the buffer belongs. Buffer ownership can be increased so that more than one task can simultaneously own a shared buffer. Special facilities are provided to assure that if a buffer is owned by more than one task, it is only returned to its pool when the slowest owner finally releases it.

The **Memory Manager** controls the dynamic allocation of memory to tasks in the multitasking environment. Multiple sections of user defined memory can be controlled by the Memory Manager. The memory in each section must be contiguous but the sections themselves do not have to be contiguous.

A task can request the Memory Manager to allocate a contiguous block of memory of any size. When finished with the block, the task requests the Memory Manager to free the memory for use by other tasks.

A particularly unique feature of the Memory Manager permits any block of memory (including those acquired from the Memory Manager) to be treated as memory from which smaller private blocks can be dynamically allocated.

The **Circular List Manager** provides a general purpose circular list facility for maintaining compact lists of 8-bit, 16-bit or 32-bit variables. Circular lists are particularly useful for managing character streams associated with input/output devices.

The **Linked List Manager** provides a fast, general purpose doubly-linked list facility for maintaining lists of arbitrary application data structures (objects).

The Linked List Manager removes the tedium and the frequent errors usually encountered when each application must manipulate the linkages of different types of objects on different lists. Objects can reside on multiple lists at the same time, a characteristic frequently encountered in real problems but ignored by most list manipulation software.

2.4 Starting AMX

An AMX operating system consists of AMX, the subset of its managers which you choose to use and your complement of application programs. All of these modules are connected together to form the AMX operating system as described in the AMX Tool Guide.

Before launching AMX, you must establish the required operating mode for the particular target processor you are using. Guidelines are presented in the corresponding AMX Target Guide.

AMX is always launched from your `main` program (or startup module) by calling application procedure `cjkslaunch` as in the following example.

```
void main(void)
{
   :
   :
   cjkslaunch();                         /* Launch AMX               */
   }
```

Source code for a typical launch procedure `cjkslaunch` is provided in AMX module `CJZZZUF.C`. It is expected that you will edit this module to provide any custom startup or shutdown processing you deem necessary.

AMX disables the interrupt system at the time you launch AMX. Interrupts remain disabled during the startup process while AMX initializes its internal parameters. AMX enables interrupts prior to calling each of your Restart Procedures.

Your AMX operating system can be launched in two ways: temporarily or permanently. By default, procedure `cjkslaunch` launches AMX for temporary use since this is the type of launch most useful during system development.

The launch type, temporary or permanent, is determined by a configuration parameter in your AMX Target Configuration Module (see Chapter 16).

Temporary Launch

Your AMX operating system can be started, allowed to run for a while and then stopped. This type of operation is called a temporary launch. The launch type is defined as temporary in your AMX Target Configuration Module.

The *main* procedure calls *cjkslaunch* to initiate the launch. The *cjkslaunch* procedure in AMX module *CJZZZUF.C* calls AMX at its entry point *cjksenter* passing it a pointer to the User Parameter Table in your System Configuration Module.

When an AMX system is launched for temporary execution, it executes until one of your application tasks calls the AMX shutdown procedure *cjksleave* requesting an orderly shutdown of the AMX system (see Chapter 3.11). The *cjksleave* procedure in AMX module *CJZZZUF.C* calls AMX at its shutdown exit point *cjksexit*.

The *cjksleave* caller can return information to the procedure that launched AMX describing the reason for the shutdown as illustrated in the following example.

```
/* Start AMX for temporary execution                                */

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

                                        /* User Parameter Table     */
extern CJ_CCONST1 struct cjxupt CJ_CCONST2 cj_kdupt;

                                        /* Application exit info    */
void        *exitinfop;                 /* A(exit information)      */
int         exitstatus;                 /* Exit status              */


int CJ_CCPP cjkslaunch(void)
{
   cjksenter(&cj_kdupt);                /* Launch AMX               */

   if (exitstatus) {
            /* Interpret <exitstatus> and <*exitinfop> to           */
            /* determine your application's reason for exit.        */
            }

   return(exitstatus);
   }


void CJ_CCPP cjksleave(
int         status,                     /* Exit status              */
void        *infop)                     /* Pointer to exit info     */
{
   exitstatus = status;                 /* Return info to launcher  */
   exitinfop = infop;
   cjksexit();                          /* Shutdown AMX             */
   }
```

Permanent Launch

In most applications, your AMX operating system is resident in ROM or loaded into RAM. AMX is started and given permanent control of the processor. The launch type is defined as permanent in your AMX Target Configuration Module.

The *main* procedure calls *cjkslaunch* to initiate the launch. The *cjkslaunch* procedure calls AMX at its entry point *cjksenter* passing it a pointer to the User Parameter Table in your System Configuration Module.

Since the launch is permanent, there is no return from *cjksenter*.

```
/* Start AMX for permanent execution                                    */

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

                                            /* User Parameter Table     */
extern CJ_CCONST1 struct cjxupt CJ_CCONST2 cj_kdupt;

void CJ_CCPP cjkslaunch(void)
{
   cjksenter(&cj_kdupt);                    /* Start AMX; never returns */
   }
```

3. Application Tasks

3.1 Task Creation

The AMX Multitasking Executive provides a simple solution to the complexity of real-time multitasking. AMX supervises the orderly execution of a set of application program modules called tasks. Each task solves a particular problem and provides a specific functional capability within the system.

The maximum number of tasks in a system is user defined in your System Configuration Module (see Chapter 15.4). The defined maximum sets an upper limit on the number of actual tasks that can be created by the application.

Tasks can be created in two ways. Tasks can be predefined in your AMX System Configuration Module which is processed by AMX at startup. Tasks defined in this fashion are automatically created by AMX but are not started. Restart Procedures and tasks can also dynamically create tasks.

AMX procedure *cjtkbuild* or *cjtkcreate* is used to create a task. AMX allocates a task and returns a task identifier to the caller. The task id is a handle which uniquely identifies the particular task. It is the responsibility of the application to keep track of the task id for future reference to the task.

When a task is created, you can provide a unique 4-character task tag to identify the task. The tag can be used subsequently in a call to *cjksfind* to find the task id allocated by AMX to the particular task.

AMX uses the information in the task definition to construct a Task Control Block (TCB) for the task. The TCB of each task is used exclusively by AMX to control task execution. At any instant in time, the content of the TCB, as maintained by AMX, completely describes the state of the corresponding task.

Tasks which do not receive messages are written as C procedures without formal parameters. These tasks must be started using AMX procedure *cjtktrigger*. For example, a task that immediately ends would appear as follows:

```
void CJ_CCPP task1(void)
{
   }
```

Tasks which must receive a message are written as C procedures which immediately call *cjmbwait* or *cjmxwait* to wait for a message from a mailbox or message exchange. These tasks must be started using the AMX procedure *cjtktrigger* and sent the necessary parameters through a *cjmbsend* or *cjmxsend* call. For example, a task which receives a message and then ends would appear as follows:

```
static CJ_ID mailboxid;

void CJ_CCPP task2(void)
{
   struct cjxmsg message;
                   /* Wait on mailbox                                */
                   /* Wait at priority 5 indefinitely                */
   cjmbwait(mailboxid, &message, 5, 0);
   }
```

AMX also supports a special type of task called a message exchange task. A message exchange task has its own private message exchange in which the task receives messages. The messages are automatically delivered to the task on the task's stack ready for processing by the task. Tasks of this type are created as described in Chapter 14.6.

3.2 Task States

A task is always in one of the following states:

Idle
Ready
Run
Wait
Halt

When a task is created, AMX assigns it a Task Control Block and sets it in the idle state. An idle task has no outstanding requests to execute pending. It is waiting to be triggered.

A ready task has an outstanding request to execute or is ready to resume execution after having been interrupted or waiting.

A task which is executing is the only task which is in the run state.

A task is in the wait state when it is blocked pending the occurrence of some event. The wait state is always qualified with an indication in the TCB as to what the task is waiting for.

The halt state is reserved for use by KADAK debug utilities to suspend all task execution for debugging purposes.

Figure 3.2-1 illustrates the task states and shows the state transitions which are possible. The halt state is not explicitly shown.

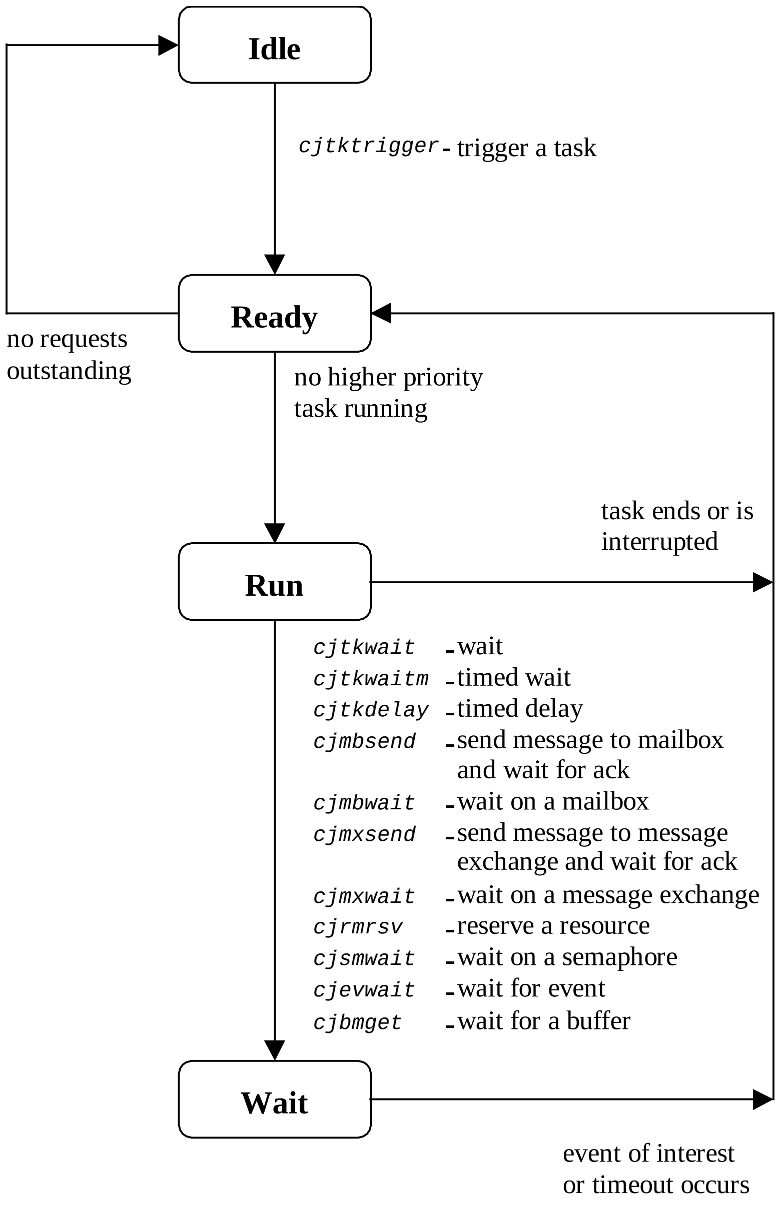

Idle
cjtktrigger - trigger a task
Ready
no requests outstanding
no higher priority task running
task ends or is interrupted
Run
cjtkwait - wait
cjtkwaitm - timed wait
cjtkdelay - timed delay
cjmbsend - send message to mailbox and wait for ack
cjmbwait - wait on a mailbox
cjmxsend - send message to message exchange and wait for ack
cjmxwait - wait on a message exchange
cjrmrsv - reserve a resource
cjsmwait - wait on a semaphore
cjevwait - wait for event
cjbmget - wait for a buffer
Wait
event of interest or timeout occurs

Figure 3.2-1 AMX Task State Diagram

3.3 Starting a Task

At startup, AMX initializes all predefined application tasks into an idle state. Once idle, a task cannot execute until AMX receives a directive to start the task. How then does an AMX system get off the ground?

Only one AMX directive is provided to start a task. The request for task execution is called a trigger. The `cjtktrigger` call is used to trigger a task.

Requests to start a task can be issued in any of the following application modules.

Restart Procedure
Application Task
Interrupt Service Procedure
Timer Procedure

The Restart Procedure provides the first opportunity to start an application task. At startup, after all predefined tasks have been created, AMX executes all of the Restart Procedures described in the Restart Procedure List. A Restart Procedure can be used to request execution of one or more tasks in the system. The AMX Task Scheduler subsequently starts the highest priority task capable of execution.

Once a task is executing, it is allowed to trigger any task in the system, including itself. AMX guarantees that the called task is executed in response to one caller at a time.

Requests for task execution can also be initiated in response to device interrupts. When an interrupt occurs, the device Interrupt Service Procedure (ISP) is executed. The ISP can issue a `cjtktrigger` call to start a particular task. Whenever an ISP requests to start a task, AMX temporarily suspends the interrupted task, thereby leaving it in a state ready to resume execution. The AMX Task Scheduler is then invoked to start (or return to) the highest priority task ready for execution. It is this type of operation that permits a task to begin execution in response to an event which is considered to be of higher priority than the task which was running.

AMX also permits tasks to be executed at periodic intervals. For example, a periodic application interval timer (see Chapter 5) can be started by your application. When the timer expires, the AMX Kernel Task executes the associated application Timer Procedure. The Timer Procedure can issue a `cjtktrigger` call to AMX to request execution of any given task.

3.4 Task Priority

Task priorities are used by the AMX Task Scheduler to determine which task to execute. At all times, the task with the highest priority which is capable of execution will run.

A task's priority is defined at the time the task is created. Task priorities range from 0 (highest) to 255 (lowest). Priorities 0 and 128 to 255 inclusive are reserved for AMX use. The AMX Kernel Task executes at priority 0 above all other tasks.

Application tasks can be assigned priorities 1 to 127 inclusive.

If more than one task is assigned the same priority, AMX will assign the tasks relative priorities according to the chronological order in which they were created with the first created task having the higher priority.

A task may change its priority or the priority of any other task. This practice is not recommended however. Experience has shown that this facility is rarely required and all too often is abused. The task's new priority remains in effect until you decide, if ever, to change it again.

3.5 Task Execution

AMX starts a task by making a procedure call to the task. AMX starts execution of a task at the task start address specified in the task's definition. The task is started in response to a request for its execution. Requests can come from Restart Procedures, tasks, Interrupt Service Procedures or Timer Procedures (see Chapter 3.3).

AMX starts the task when no tasks of higher priority are capable of execution. A task can therefore only execute if all higher priority tasks are idle or suspended for some reason.

When AMX starts the task, the following conditions exist:

Interrupts are enabled.
All registers are free for use.
The task stack is in use.

Once a task begins execution, it has complete control of the processor. The interrupt system must be left enabled to permit device interrupts to be serviced. If a task must disable interrupts for some reason, it is recommended that this period be kept as short as possible so that the system's interrupt response time is not degraded.

The application task can execute without concern for the fact that interrupts will occur and will be serviced. If higher priority tasks become ready for execution, the task will be suspended temporarily by AMX. When higher priority tasks become suspended or have completed their operation, the interrupted task will be permitted by AMX to resume execution from its point of interruption.

Occasionally a task must perform a sequence of operations without interference by other tasks. If the sequence is too long to permit interrupts to be disabled, the task can request AMX to become temporarily privileged. During the period of privileged execution, interrupts remain enabled but execution of any higher priority task, including the AMX Kernel Task, is inhibited. Privileged operations should be kept as short as possible.

3.6 Task and Event Synchronization

AMX offers several simple forms of task/event synchronization.

Using the `cjtkwait` call, a task can unconditionally **wait** for an event. The event can be task dependent, device dependent or time dependent. The task, having issued a `cjtkwait` call, remains suspended unconditionally until another task, an Interrupt Service Procedure or a Timer Procedure issues a `cjtkwake` call requesting AMX to **wake** up the particular waiting task. This `cjtkwait/cjtkwake` pair can be used to provide simple event synchronization.

AMX also supports this form of synchronization with an automatic timeout facility. A task can issue a `cjtkwaitm` call specifying the maximum interval which the task is willing to wait for the event to occur. If no other task, ISP or Timer Procedure issues a `cjtkwake` call in the interim, AMX will automatically wake up the task when the interval expires. The task receives from AMX an indication whether or not a timeout occurred.

This synchronization facility offers the advantage of simplicity. The mechanism is an inherent part of the AMX kernel. The disadvantage is that the event signaller must know the task id of the task which is waiting.

The **Mailbox Manager** and the **Message Exchange Manager** allow a task to be synchronized to another task using their **call and wait** message passing facilities. A task sends a message to a mailbox or a message exchange using the AMX `cjmbsend` or `cjmxsend` call and waits for acknowledgement. The receiving task must use the AMX procedure `cjmbwait` or `cjmxwait` to wait for a message at the mailbox or message exchange. When the receiving task eventually receives the sender's message, it can issue a `cjtkmsgack` call to wake the calling task. The sender will then resume execution knowing full well that its message has already been received by the task to which it was sent. If the receiving task does not issue a `cjtkmsgack` call, AMX will do so automatically when the receiving task ends.

The **Semaphore Manager** provides counting semaphores with queuing and timeout facilities for mutual exclusion and resource management. It also offers a unique resource semaphore which extends to semaphores the concept of ownership of the corresponding resource.

The **Event Manager** offers the best solution for complex event coordination. It permits a task to be synchronized to any or all of a particular set of events. It also allows more than one task to wait for the same events.

Each of the synchronization methods provided by the managers share several common features. In each case, the signaller does not need to know the identity of task(s) waiting for its signal. When multiple tasks wait on a semaphore, event group, mailbox or message exchange, each task specifies the priority at which it wishes to wait. Finally, any task waiting on a semaphore, event group, mailbox or message exchange can specify the maximum interval which the task is willing to wait.

3.7 Task Timing

The AMX Clock Handler and Kernel Task act as a Timer Manager providing timing facilities for use by tasks. It has been shown in Chapter 3.6 that tasks can wait for an event to occur with an automatic timeout. The task is suspended following its wait request until the event occurs or the interval specified in the call expires.

The *`cjtkdelay`* call to AMX can also be used by a task to implement a delay. The delay interval is specified in system ticks. The task is suspended until the interval expires. AMX assures that the task automatically resumes execution, provided that no higher priority task is able to execute.

Other timing functions required by the task can be implemented using interval timers (see Chapter 5). Timers are 32-bit counters. Timing resolution is in multiples of the system tick. The AMX routine *`cjtmconvert`* is available to convert milliseconds to the equivalent number of system ticks.

Timers can be created at any time by a call to the AMX routine *`cjtmcreate`*.

A timer is started by writing the timer interval to it using the AMX procedure *`cjtmwrite`*. At any instant, a task can read the time remaining in the interval by calling AMX routine *`cjtmread`*. A timer can be stopped by writing zero to it. A timer can be deleted when it is no longer required by a call to *`cjtmdelete`*. These simple procedures give the task complete control over interval timing.

Whenever an interval timer expires, AMX executes an associated Timer Procedure. Using this feature, a task can start an interval timer and rest assured that, when the interval expires, the action determined by the associated Timer Procedure will be performed. Since the Timer Procedure is called by the AMX Kernel Task which has the highest priority in the system, the Timer Procedure executes at a priority higher than that of any application task.

Interval timers must be used by tasks wishing to measure time. Instruction counting loops are of no value in a multitasking system. Since a task is being constantly interrupted and occasionally suspended to execute higher priority tasks, any timing performed by counting instructions within a program loop will be in error.

3.8 Ending a Task

When a task completes its appointed function, it must relinquish control of the processor to AMX. The AMX Task Scheduler will then give control of the processor to the next highest priority task ready to execute.

AMX starts a task by a procedure call to the task at the task start address. The task program is, therefore, a procedure. When the task is finished, it returns to AMX in normal end-of-procedure fashion.

A task may wish to abort execution under error conditions. The task's stack may be deeply nested when the error condition arises. To allow a task to terminate under these circumstances, AMX provides the *`cjtkend`* exit facility. Any task which calls *`cjtkend`* is immediately terminated by AMX.

A task executes once for every trigger request which AMX receives to start the task. When the task ends as just described, AMX examines the task to determine if any additional requests for execution of the task are pending. If outstanding trigger requests are present, AMX flags the task as ready to run again. The AMX Task Scheduler will then immediately start this task again. This process continues until the task finally ends and no requests for execution of the task are present. At that time, AMX places the task into the idle state.

If the task has retrieved a message from a mailbox or message exchange, AMX must perform some additional processing when the task ends. AMX checks to see if the message came from a task. If so, AMX tests to see if the sender is waiting for an acknowledgement of receipt of its message. If the sender is waiting, AMX sets it into the ready state ready to resume execution.

It is possible that an application task may never end. Such a task, once started, runs forever. For example, a task might wait for an event and then do some event-dependent processing. Once the processing is complete, the task waits for the next event.

In this example, the task never reaches a logical end. Note, however, that the task does become suspended awaiting an event. A task which has no logical end and which never suspends itself is said to be compute bound.

Even if a task does wait, it is still possible that the task may effectively be compute bound. For instance, assume that a low priority task repetitively sends a message to a higher priority task and waits for an answer. Also assume that the higher priority task will always provide an immediate response. In this case, the lower priority task will always be allowed to resume after its message is sent even though a temporary suspension does occur. The task will block all lower priority tasks.

Note

A compute bound task inhibits execution of all lower priority tasks.

3.9 Message Passing

AMX supports the passing of a message to a task through a mailbox or message exchange.

Messages can be sent to a task by:

> Application Tasks
> Interrupt Service Procedures
> Timer Procedures
> Restart Procedures
> Exit Procedures

You can send a message to a mailbox or message exchange using the AMX procedure `cjmbsend` or `cjmxsend` respectively. The maximum number of parameter bytes that can be sent in a message is configured by you in your System Configuration Module. The parameter bytes are completely application dependent.

Messages must be integer aligned. In the following examples, it is assumed that your C compiler aligns arrays and structures on boundaries that are integer aligned.

```
int       i;
short int iarray[6];
char      carray[12];
struct msg {
          char         m1;
          char         m2;
          short int    m3;
          long         m4;
          } msga;
```

The following calls will send the specified message to the mailbox with mailbox id `mbid` or to the message exchange with exchange id `mxid` without waiting for a response:

```
wack = CJ_NO;                          /* Do not wait              */
cjmbsend(mbid, &i, wack);              /* Send i                   */
cjmbsend(mbid, iarray, wack);          /* Send iarray              */
cjmxsend(mxid, carray, wack, 0);       /* Send carray at           */
                                       /* message priority 0       */
cjmxsend(mxid, &msga, wack, 3);        /* Send msga at             */
                                       /* message priority 3       */
```

The messages are stored in FIFO order in a mailbox or message exchange message queue. The caller specifies the priority level of the message when sending to a message exchange. The priority level (0, 1, 2 or 3) determines the message queue at the message exchange into which the message will be placed.

If the caller is a task, it can request AMX to suspend its operation until the task receiving the message acknowledges its receipt of the message. To wait for acknowledgement, the task sending the message sets `wack` non-zero in its `cjmbsend` or `cjmxsend` call.

AMX uses message envelopes for parameter transmission. AMX gets a free envelope, moves the caller's message parameters into it and adds the envelope to the mailbox or message exchange message queue.

If the sender requested to wait, AMX inserts information into the envelope which allows AMX to remember that the sending task has been suspended waiting for acknowledgement of the receipt of its message by some other task.

Note that when AMX returns from a call to send a message, the memory occupied by the caller's message is available for reuse by the caller.

Note

AMX always copies *`CJ_MAXMSZ`* bytes from the sender's message into the envelope.

Consequently, the transmitted message will unconditionally be *`CJ_MAXMSZ`* bytes in length even if your message is a single integer.

Figure 3.9-1 provides an example of the manner in which messages are allowed to queue on a message exchange. In the example, the following situation is assumed to exist at the message exchange on which the destination task is waiting.

The message exchange has four message queues corresponding to the four priority levels at which it can post messages. Level 0 is the highest priority; level 3 is the lowest priority. The example shows that no messages are pending on levels 0 or 2. Three messages, I, J and K, are pending on level 1. Two messages, L and M, are pending on level 3. One message, X, is currently being serviced by the destination task.

The destination task has been interrupted and it is ready to resume execution. As a result of the interrupt, task YY was allowed to execute because it was of higher priority than the destination task. Task YY made a request to AMX to send a message to the message exchange at message priority level 1. As a result of this request, AMX moved the message parameters from task YY into a message envelope and added this envelope to the bottom of the level 1 message queue associated with the message exchange. The result is shown in Figure 3.9-1.

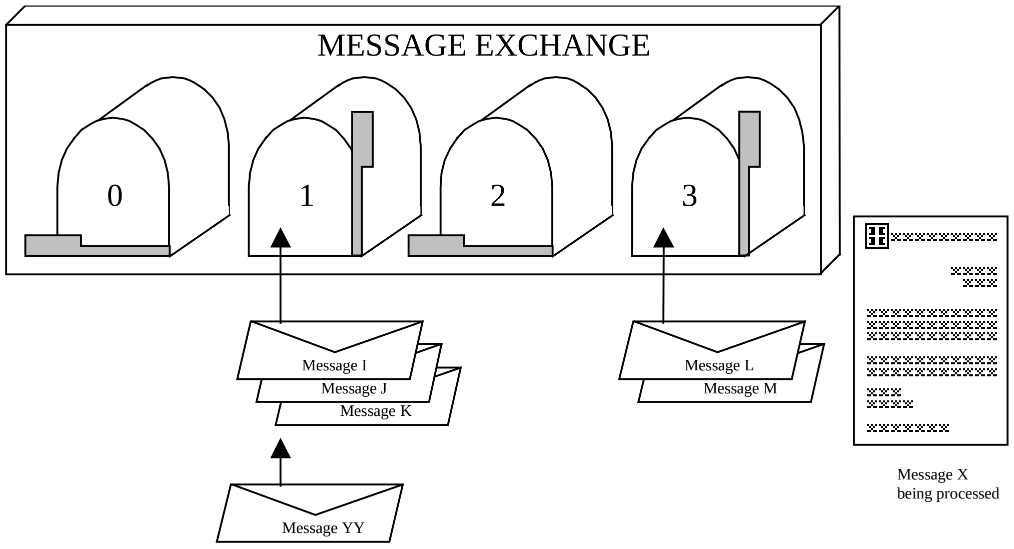


Figure 3.9-1 Message Transmission

In due course the destination task will continue to execute and complete its processing of the current message, X. The destination task will then request AMX for a message from the message exchange three more times in succession to process messages I, J and K. Finally, the message from task YY will be retrieved from the message exchange and processed.

Figure 3.9-2 illustrates the situation at the destination task at the instant it begins to process the message from task YY. Provided that no additional calls to post messages to the message exchange have occurred, the messages will be as shown. Message queues 0, 1 and 2 are empty. Messages L and M are still pending at priority level 3. The message from task YY is ready for processing by the destination task.

When the destination task retrieves a message from the message exchange, the message parameters are already removed from the envelope and copied to the storage area provided by the task, usually on the task's stack. The message parameters received by the destination task are in exactly the same order as they were sent by the caller.

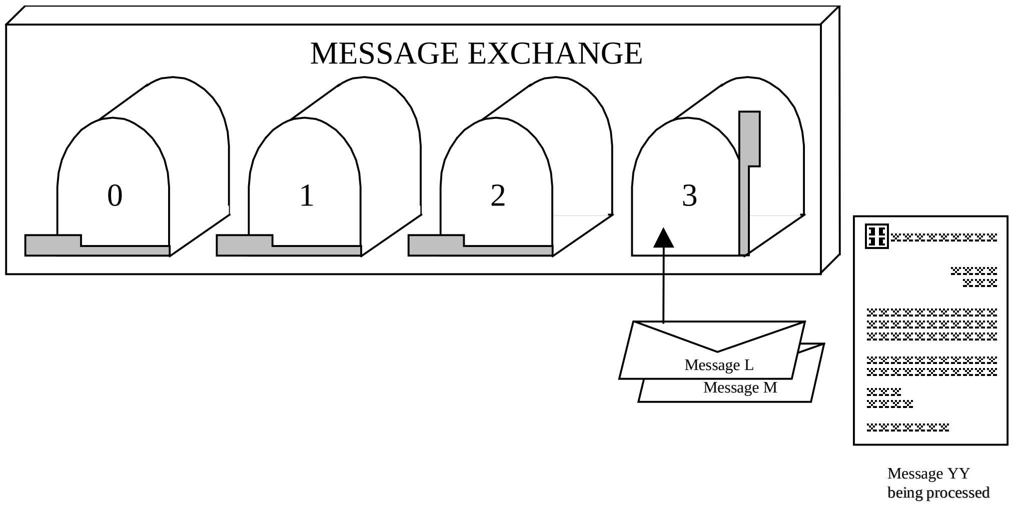


Figure 3.9-2 Message Reception

It is important to note that a copy of the sender's message parameters is sent to the destination task. Once the sender's parameters have been copied into the message envelope, the caller is free to reuse the parameter storage if desired. Thus, as soon as procedure *cjmbsend* or *cjmxsend* returns to its caller, the parameter variables are free for reuse.

The destination task must provide a pointer to storage large enough to hold the parameters it is to receive. Since AMX unconditionally delivers messages of *CJ_MAXMSZ* bytes in length, the message storage must be of at least that size.

The AMX task which receives a variety of messages should declare a union *rmsg* in order to reference the different messages. Note that the union must include one instance of structure *cjxmsg* to ensure that *rmsg* is large enough to hold any AMX message.

```
   union rmsg {
            struct cjxmsg amxmsg;
            char      c;
            int       i;
            short int iarray[6];
            char      carray[12];
            struct msg msga;
            };
```

Then the task can be written as follows:

```
static CJ_ID mxid;

void CJ_CCPP taskn(void)
{
   union rmsg msg;

   cjmxwait(mxid, &msg, 0, 0);           /* Wait at priority 0 forever*/
   :
   :
   The received message can now be accessed from union msg
   :
   }
```

In this simple example, the task receiving the messages has no obvious way of determining how to interpret the message. Is the message one character (*msg.c*) or a whole structure (*msg.msga*)? This dilemma is usually solved by including an application specific operation code with all messages.

Note that we have declared the task's received message as a union and then used variable *msg* to access the parameters in the message. AMX passes parameters by value with no concern for the restrictions imposed by various high level languages. Hence a whole array or structure up to a maximum of *CJ_MAXMSZ* bytes can be passed by value by AMX as a message.

3.10 Restart Procedures

The manner in which the operating system begins execution is application dependent. Execution begins in the user domain providing the opportunity for hardware specific and application dependent setup prior to the initialization of the AMX system. For example, hardware interfaces may require custom configuring. In some systems, it might be desirable to perform a memory integrity check before system startup is permitted.

AMX disables the interrupt system at the time you launch AMX. Interrupts remain disabled during the startup process.

Once AMX has initialized all of its internal variables and structures, it enables interrupts and executes the sequence of application Restart Procedures provided in the Restart Procedure List in your System Configuration Module (see Chapter 15.7). These procedures can invoke AMX services to start tasks and initialize interval timers.

When AMX calls a Restart Procedure, the following conditions exist:

Interrupts are enabled.
All registers are free for use.
The AMX Kernel Stack is in use.

Restart Procedures are written as C procedures without formal parameters.

```
void CJ_CCPP rrproc(void)                /* Restart Procedure        */
{
   :
   Do restart processing
   :
   }
```

Restart Procedures can enable specific device interrupts if required. Note that interrupts from a device should not be enabled until the application ISP has been installed and made ready to handle the interrupting device.

A Restart Procedure must not issue any AMX directives which would in any way suspend or terminate task execution.

Note

Restart Procedures must only use AMX services which are marked in Chapter 18 as
■ Restart Procedure

Restart Procedures use the AMX Kernel Stack. In addition to the minimum stack size required for the AMX Kernel Stack, you must allocate sufficient stack to satisfy the worst case requirements of all application Restart Procedures.

3.11 Exit Procedures

An AMX system can be shut down in an orderly fashion by a task call to procedure *cjksleave*. The manner in which the operating system ends execution is application dependent. For example, hardware interfaces may require restoration to their initial states.

AMX supervises the shutdown process by sequentially calling all of the application Exit Procedures in the order defined in the Exit Procedure List in your System Configuration Module (see Chapter 15.7). Once all Exit Procedures have been executed, AMX returns to the original program that launched AMX.

When AMX calls an Exit Procedure, the following conditions exist:

> Interrupts are enabled.
> All registers are free for use.
> The stack is the task stack which was in effect when *cjksleave* was called.

Exit Procedures are written as C procedures without formal parameters.

```
void CJ_CCPP epproc(void)                 /* Exit Procedure            */
{
   :
   Do exit processing
   :
   }
```

Exit Procedures execute in the context of the task which issued the *cjksleave* call. They are therefore free to use all services normally available to tasks. For instance, an Exit Procedure could use *cjmbsend* to send a shutdown message via a mailbox to a task and wait for that task to do its shutdown processing. When the Exit Procedure resumes after the *cjmbsend* call, it returns to AMX which then calls the next Exit Procedure in the list.

An Exit Procedure must not issue any AMX directives which would in any way terminate task execution.

Note

Exit Procedures must only use AMX services which are marked in Chapter 18 as
■ Exit Procedure

Exit Procedures use the stack of the task which called *cjksleave*. It is therefore essential that the task which calls *cjksleave* be allocated sufficient stack to satisfy the worst case requirements of all application Exit Procedures.

3.12 Task Enhancements

AMX offers several task enhancements which, although rarely used, can occasionally come in handy. These enhancements are briefly summarized below.

Task Control Block Extension

Within a task's Task Control Block, 16 bytes are reserved for the private use of the task. Their use is completely determined by the application.

A Task Control Block for a particular task can be located using procedure *`cjtktcb`*. The private storage is located in structure member *`xtcbuser`* in the Task Control Block.

You must use structure definition *`cjxtcbs`* from header file *`CJZZZSD.H`* to access this storage.

Stack Fences

When AMX creates a task, it installs a fence at the top and bottom of the task's stack. The fence is a 32-bit value containing the unusual pattern *`0x55555555`* ('*`UUUU`*').

The 4-character task tag is copied to the stack immediately below the top fence as a debugging aid to identify each stack's owner.

These fences can be of significant help during the development and testing of your AMX system. When you are trying to detect a fault in your system, it is often advisable to examine your task stacks carefully to be sure that none of the fences have been altered. An altered fence is an indication that you have a task which is overflowing or underflowing its stack or, worse yet, a bad piece of code which is writing to some task's stack in error.

4. Interrupt Service Procedures

4.1 The Processor Interrupt Facility

The key to event-driven, real-time, multitasking systems is the processor's interrupt facility. Tasks execute with the interrupt facility enabled permitting the system to respond to a real-time event.

The hardware interrupt mechanism is an automatic facility provided by the processor. AMX permits the system designer to determine how the hardware interrupt facility will be employed.

Tasks must execute with the interrupt facility enabled.

From time to time, AMX must inhibit interrupts while it performs a critical, indivisible sequence of operations. AMX keeps such intervals very short. For instance, even while AMX is switching from one task to another, it is able to respond to interrupts.

To further improve interrupt response, AMX permits nesting of interrupts on processors which support this feature. As soon as the interrupt request has been cleared, interrupts can be enabled to permit response to other external events.

When an interrupt occurs, the processor saves enough processor dependent information to permit the processor to eventually resume the interrupted process. The manner in which this is done is of course processor dependent. In some cases, a processor interrupt mask is set to *n* thereby disabling all external interrupts of priority less than or equal to *n*. On some RISC processors, this ability to nest interrupts is provided by AMX microcode.

Hardware external to the processor usually identifies the interrupt source. The processor uses the interrupt identifier in a processor dependent fashion to find the unique device dependent pointer. Program execution resumes at the address specified by this pointer. The program located at this address is called an Interrupt Service Procedure (ISP).

In general, an ISP saves the registers it wishes to use, services the device, restores the registers, enables the interrupt and returns to the executing program at the point of interruption.

If more than one device is connected to the same external interrupt source, your ISP must determine the interrupt source in one of several ways. The simplest, but slowest, is a software poll of the devices. The ISP tests each device sequentially to determine the source of the interrupt and branches to a device service procedure to handle the specific interrupt.

Alternatively, external hardware can be added to provide unique vectoring for each device which can generate interrupts. If this approach is adopted, then a separate ISP must be provided for each device.

The AMX Interrupt Supervisor simplifies ISP operation within the AMX multitasking environment. The AMX Interrupt Supervisor permits an ISP to communicate with any task in the system. The remainder of this chapter describes how ISPs are used within a system.

4.2 ISPs for External Interrupts

Two types of ISPs exist: nonconforming ISPs and conforming ISPs.

Nonconforming ISPs

A nonconforming ISP must quickly service the device to remove the interrupting condition. The ISP must preserve all registers which it uses. The nonconforming ISP cannot make calls to any AMX service procedures.

The nonconforming ISP executes in the context of the process (task, ISP, AMX kernel) which was interrupted. The ISP therefore uses the stack of the interrupted process. Consequently, all stacks must be large enough to meet the worst case needs of all nonconforming ISPs.

The nonconforming ISP can enable interrupts to allow interrupt service by other higher priority nonconforming ISPs. The nonconforming ISP **MUST NOT** allow an interrupt to a conforming ISP.

Conforming ISPs

A conforming ISP is intended for use with AMX. A conforming ISP consists of an ISP root and an Interrupt Handler. The processor vectors to the ISP root which informs the AMX Interrupt Supervisor that the interrupt has occurred. The Interrupt Supervisor preserves the state of the interrupted task and, if necessary, switches to an interrupt stack. The Interrupt Supervisor then calls the associated Interrupt Handler. Your handler must perform all services required by the device. Your handler is free to make procedure calls to a subset of the AMX service facilities.

The conditions that will exist upon entry to your Interrupt Handler are dependent upon the target processor, the tool set which you are using and the language in which your Interrupt Handler is programmed. In general:

Interrupts are disabled (masked) at a device dependent priority level.
A subset of the registers are free for use.
The AMX Interrupt Stack is in use.

When your Interrupt Handler executes, external interrupts of priority less than or equal to the interrupting device are always disabled. Your Interrupt Handler must NOT enable any interrupts of lesser or equal priority. Your handler can enable higher priority external interrupts if permitted by the target processor.

When device service is completed, your Interrupt Handler returns to the ISP root which informs the AMX Interrupt Supervisor that interrupt service is complete.

AMX monitors calls made by the Interrupt Handler to AMX service procedures. If no such calls have been made, the AMX Interrupt Supervisor automatically restores the state of the interrupted task and allows the ISP root to return directly to the interrupted task at its point of interruption.

If the Interrupt Handler requested AMX to initiate or resume execution of some task of higher priority than the interrupted task, the AMX Interrupt Supervisor suspends the interrupted task and marks it as ready to resume execution at the earliest opportunity. The AMX Task Scheduler is then invoked to determine the highest priority task capable of execution.

If interrupts nest, the Interrupt Supervisor defers its task switching checks until all of the concurrent interrupts have been serviced.

In due course, AMX returns to the ISP root ready to resume execution at the point of interruption. It is **important** to note that, because of task switching invoked by the AMX Interrupt Supervisor, there may be a significant delay before AMX returns to the ISP root and resumes execution of the interrupted task.

Note

Interrupt Handlers must only use AMX services which are marked in Chapter 18 as
■ ISP

The Interrupt Handler is free to use the following AMX service procedures to communicate with tasks.

cjmbsend	Send a message to a mailbox
cjmxsend	Send a message to a message exchange
cjtktrigger	Request a task to execute
cjtkwake	Wake a task known to be waiting for this interrupt
cjsmsignal	Signal to a semaphore
cjevsignal	Signal one or more events in an event group

Interrupt Handlers are also free to use the following AMX buffer management and timing facilities.

cjbmget	Get a buffer
cjbmfree	Free a buffer
cjbmuse	Add to buffer use count
cjtmwrite	Start/stop an interval timer
cjtmread	Read an interval timer

The full range of AMX circular list handling routines (see Chapter 12) can be used by application Interrupt Handlers. These circular lists can be especially useful to provide character buffering. For example, an input device Interrupt Handler can add characters to the bottom of a circular list while a related task removes them from the top of the list.

The AMX List Manager services (see Chapter 13) can also be used by Interrupt Handlers. Note that an Interrupt Handler should not manipulate keyed lists.

Conforming ISP Construction

The construction of an Interrupt Service Procedure (ISP) to service an external interrupt is a simple process.

The conforming ISP consists of two parts: the ISP root and your Interrupt Handler. The **ISP root** is a code fragment generated automatically for you in your Target Configuration Module by the AMX Configuration Generator (see Chapter 16). A short text file called a Target Parameter File is edited to define the set of conforming ISPs you wish to construct. For each ISP, you provide the name of the corresponding Interrupt Handler which will service the device. The AMX Configuration Generator is then used to convert your Target Parameter File into an AMX Target Configuration Module containing your customized application ISP root.

The **Interrupt Handler** can be written in assembly language or C. Use assembly language if speed of execution is critical. Follow the guidelines provided in the AMX Target Guide for your target processor.

Interrupt Handlers can be written as C procedures with or without a single 32-bit formal parameter. The following example of a device Interrupt Handler illustrates how little application code must be programmed to satisfy AMX. The example in Chapter 4.7 shows an Interrupt Handler which receives an application dependent parameter.

```
void CJ_CCPP deviceih(void)
{
   local variables, if required
   :
   Clear the source of the interrupt request.
   Perform all device service.
   :
   }
```

Note

The conforming ISP root is created by the AMX Configuration Generator in your AMX Target Configuration Module.

4.3 Nested Interrupts

AMX supports nested interrupts on those processors which provide this feature. The AMX Interrupt Supervisor maintains a private nesting level indicator. AMX is informed of the start and end of each interrupt by the ISP root.

When AMX sees that a task has been interrupted, it switches to a predefined Interrupt Stack. If AMX detects that the interrupt has occurred during execution of a device Interrupt Service Procedure, then no stack switching occurs. The interrupted ISP is suspended and the new ISP is started.

When an ISP ends, AMX takes action based on the state of its nesting indicator. When a nested interrupt ISP is completed, AMX returns to the interrupted ISP. When the last interrupt ISP (corresponding to the first task interrupt) is completed, AMX prepares to return to the interrupted task. If, as a consequence of interrupt service, a significant event has been declared, AMX suspends the interrupted task and goes to the AMX Task Scheduler to find the highest priority task which is ready to execute.

Since the AMX Interrupt Stack is used for nesting interrupts, the stack size must be large enough to support the worst case combination of nested ISPs. Each nested interrupt requires that the Interrupt Stack be increased to meet the needs of the additional nested ISP.

Warning

You must not permit a conforming ISP to interrupt a nonconforming ISP.

The AMX clock ISP is a conforming ISP.

4.4 ISP/Task Communication

AMX provides a set of service procedures to ease the communication between tasks and device Interrupt Handlers. These AMX procedures simplify event synchronization and permit parameter passing to tasks.

Wait/Wake Synchronization

The *cjtkwait/cjtkwake* pair of procedures is often used for event synchronization. A task issues a *cjtkwait* call to AMX to wait unconditionally for an event. When the event occurs, as indicated by an interrupt, the Interrupt Handler makes a *cjtkwake* call identifying the task which it wishes to wake up.

Synchronization capability can be further enhanced using the automatic timeout facility provided by AMX. The task issues a *cjtkwaitm* call indicating the maximum interval for which it is willing to wait for the event. When the interrupt occurs, the Interrupt Handler issues the *cjtkwake* call to wake up the task. The task resumes execution with an indication provided to it by AMX that the event did occur. If, on the other hand, the interrupt does not occur within the specified timeout interval, the AMX Kernel Task will wake up the task. In this case, the task resumes execution with an indication provided by AMX that a timeout occurred. When the task resumes execution, it is, therefore, capable of determining if the event occurred within the expected interval.

Semaphore Synchronization

The AMX Semaphore Manager provides an even more powerful synchronization capability. It provides the automatic timeout facility and also allows more than one task to wait for the same event, with each task determining its own waiting priority. Furthermore, the Interrupt Handler need not know the identity of the waiting tasks (if any) or their chosen waiting priorities. The AMX Semaphore Manager will automatically ensure that the task with the highest waiting priority is given access to the semaphore first.

Synchronization using counting semaphores with an initial count of zero is achieved as follows. A task issues a *cjsmwait* call indicating the maximum interval for which it is willing to wait for the semaphore and its chosen waiting priority. When the interrupt occurs, the Interrupt Handler issues a *cjsmsignal* call to wake up the task with the highest waiting priority that is blocked on the semaphore. The task will always know when it resumes execution whether the event actually occurred or whether the maximum wait interval elapsed.

Event Group Synchronization

The AMX Event Manager provides another form of synchronization. It allows more than one task to be waiting for a specific event or for a combination of events. It also provides the automatic timeout facility. The Interrupt Handler does not have to know which tasks (if any) are waiting for its event.

Synchronization using an event group is achieved as follows. A task clears an event flag in an event group and then issues a *cjevwait* call indicating the maximum interval it is willing to wait for the event to occur. The Interrupt Handler issues a *cjevsignal* call to set the particular event flag for which it is responsible. The Event Manager then allows all tasks which are waiting for that particular event flag to be set to resume execution. The task will always know when it resumes execution whether the event actually occurred or whether the maximum wait interval elapsed.

Mailbox and Message Exchange Synchronization

An Interrupt Handler can communicate with a task by sending a message to a mailbox or message exchange. The AMX Mailbox Manager and Message Exchange Manager allow more than one task to wait for a message from a mailbox or message exchange with each task determining its own wait priority and maximum wait interval. An Interrupt Handler does not need to know the identity of the waiting tasks (if any) or their chosen waiting priorities.

Synchronization via a mailbox or message exchange is achieved as follows. A task calls *cjmbwait* or *cjmxwait* to wait on a particular mailbox or message exchange, respectively, indicating the priority of its request and the maximum interval it is willing to wait for a message to arrive.

When the interrupt occurs, the Interrupt Handler calls *cjmbsend* or *cjmxsend* to send a message to the mailbox or message exchange. The Interrupt Handler must not wait for acknowledgement of receipt of the message. The Mailbox Manager or Message Exchange Manager delivers the message to the task and allows the task to resume execution. If no interrupt occurs before the timeout interval expires, the AMX Kernel Task will force the task to resume execution with a timeout indication.

For example, a control panel might be used by an operator to initiate actions within a system. An interrupt is generated when the operator depresses a pushbutton requesting a specific function. In response to the interrupt, the Interrupt Handler reads a command register at the control panel to determine the action to be taken. Data, such as set-point settings, would also be read by the Interrupt Handler.

The Interrupt Handler interprets the command, determines the mailbox on which the task in the system responsible for performing the requested function is waiting and issues a *cjmbsend* call to send a message to that task. The data retrieved from the control panel is transmitted to that task as parameters in the message.

Chapter 3.9 provides a complete description of the parameter passing process.

Task Triggering

An Interrupt Handler can communicate with a task by invoking the task's execution. When an interrupt occurs, the Interrupt Handler issues the `cjtktrigger` call to AMX identifying the task which it wishes to have executed.

This technique can be very useful for handling slowly occurring events. For example, a device generates an interrupt and the Interrupt Handler responds by acquiring a block of data from the device. The data is stored in memory by the Interrupt Handler for subsequent use by a task. The Interrupt Handler then starts the task with the `cjtktrigger` call. It is assumed that timing is such that the task will be able to completely process the data prior to the next occurrence of a similar event. This constraint is typical in many real-time system implementations.

4.5 Task Error Traps

Many processors automatically detect the occurrence of execution errors such as division by zero, arithmetic overflow or array bound violation. These errors, by their very nature, must be handled by the application in the context of the task in which they occur. The detectable errors and the manner of detection is processor dependent. See the AMX Target Guide for the processor of interest.

AMX offers tasks a convenient method of trapping such errors. For each task, AMX maintains a pointer to a task trap handler for each detectable error.

Normally, when a task is running, AMX treats these errors as fatal if they occur. A task can specify a trap handler for a particular error trap by calling AMX service procedure *`cjksitrap`*. Thereafter, if the error occurs while running in the context of the task, the task automatically branches to its trap handler.

Note that the actual processor error exception is completely serviced by AMX. The task simply appears to have suddenly jumped to its trap handler.

AMX maintains unique trap handlers for each task. If a task is suspended and another task executes, AMX selects the trap handlers for the new task. When the suspended task resumes, its unique trap handlers are reinstated.

A task trap handler for a particular error trap is reset (cancelled) with a request to AMX to set the pointer to *`CJ_NULLFN`*.

When a task is first created, AMX sets the task's trap handlers to *`CJ_NULLFN`* forcing errors to be fatal until the task specifies its own trap handlers. Once a task defines its trap handlers, they remain defined even if the task ends execution. If the task is subsequently executed again, its previously defined trap handlers remain in effect unless altered by the task.

If a trapped error exception occurs in a task with no trap handler or in a Restart Procedure, an ISP or a Timer Procedure, AMX initiates a fatal exit as described in Chapter 14.1.

The task trap handler can be written as a C procedure with formal parameters. Since the errors are target specific, you must refer to the AMX Target Guide for the processor of interest to determine how the task trap handler must be coded.

Since the task trap handler executes in the context of the task, the task's stack must account for the stack used by the handler. An additional `sizeof(struct cjxregs)` bytes of stack is required to accommodate the processor dependent stack frame generated by AMX prior to its call to the trap handler.

In general, AMX provides the address of the fault and the state of each processor register at the time of the fault.

The register values can be examined and modified with care. If necessary, the fault pointer can be modified, with care, to force resumption at some other location in the task code. If the trap handler returns to AMX, execution will resume at the location specified by the fault pointer with registers set according to the values determined by the trap handler.

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

Note

Task trap handlers are NOT Interrupt Handlers even though on some processors the error is detected via a processor interrupt or exception.

4.6 Non-Maskable Interrupt

Most processors provide a non-maskable interrupt (NMI). This interrupt cannot be inhibited by software.

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 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.

The NMI ISP must be a nonconforming ISP. The NMI ISP cannot use AMX services. Consequently the non-maskable interrupt cannot be used as an additional, general purpose device interrupt.

Some hardware assisted debuggers may use the NMI interrupt to signal a breakpoint.

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 call any AMX service procedures.

4.7 Special Interrupts

Nonconforming Interrupts

In some systems there may be devices which generate interrupts requiring no communication or synchronization with any task in the system. For example, a high-speed scanner can interrupt the processor whenever new data readings are available. The ISP reads the new data and stores it in memory for subsequent use by any module requiring the information. These modules always use the most recently available value as seen in memory. Interrupts such as this do not need to use AMX services. The ISP simply saves the registers it requires for its own use, processes the interrupting device, restores the registers and immediately returns to the point of interruption.

Since ISPs of this type use the stack in effect at the time of the interrupt, care must be taken to assure that ALL stack sizes, including the AMX Interrupt Stack and Kernel Stack, are increased to meet the needs of the special ISPs.

ISPs of this type are called **nonconforming ISPs**. You must arrange in hardware that all such nonconforming interrupt sources are of higher priority than all conforming AMX Interrupt Service Procedures. Note that the AMX clock ISP (see Chapter 5.2) is considered to be a conforming ISP.

Occasional Task Interaction

In some applications, it may be desirable to bypass AMX for all but certain critical interrupts. For example, in a communication system, normal receive interrupts simply insert the received character into an already available input buffer. Transmit interrupts simply transmit the next available character from an output buffer. These interrupts can be serviced very quickly in a nonconforming ISP, bypassing AMX entirely.

However, when the receiver finally has a complete message or when the transmit buffer goes empty, the device service procedure must send a message to a mailbox to inform a related task that a significant event has occurred on the communication line.

The device service procedure must suddenly become a conforming AMX Interrupt Service Procedure. It can do this by restoring all saved registers and calling a conforming ISP root. The ISP root informs AMX that the interrupt has occurred and calls the device's Interrupt Handler which can then send its message to the mailbox.

Note that if interrupts of this type are employed, all AMX and task stacks must meet the requirements of the nonconforming device ISP.

Shared Interrupt Handlers

Occasionally a single Interrupt Handler can be used to service more than one identical device. For example, an Interrupt Handler for an asynchronous serial I/O device (UART) could be used to service the UART for each of several communication lines.

For the Interrupt Handler to be shared, the code must be reentrant. This usually implies that the information unique to each UART (such as the device port address and line status) must be in a data structure accessible by the handler.

A shared Interrupt Handler must therefore be able to determine which of several device dependent data structures it should use to service a particular device interrupt. AMX solves this problem by allowing an Interrupt Handler to receive a 32-bit, device dependent parameter.

The following example illustrates how little application code must be written to create a shared Interrupt Handler to handle two devices.

```
struct dvcblock {
   int      port;                        /* Device port              */
   int      line;                        /* Logical line number      */
   :
   Other dynamic device parameters
   };

struct dvcblock dcbA = {0xF8, 1};
struct dvcblock dcbB = {0xE8, 2};

void CJ_CCPP deviceih(struct dvcblock *dcbp)
{
   :
   Service device specified by pointer dcbp
   :
   }
```

To add the device ISPs to your system, you edit your Target Parameter File to declare that each of the two device ISPs uses the same Interrupt Handler `deviceih`. For each device ISP, you provide the name of the unique, public, device dependent data structure (`dcbA` or `dcbB`) to be used by the device Interrupt Handler. Then you use the AMX Configuration Generator to convert your Target Parameter File to an AMX Target Configuration Module as described in Chapter 16.

4.8 Fatal Exception Traps

Most processors detect critical faults which must be considered fatal in a real-time multitasking system. These faults often include but are not limited to:

Bus error
Address error
Privilege violation

The action taken by the processor when a critical fault is detected varies among processors. Most processors generate a unique type of interrupt called an exception or trap. Within the AMX framework, all such faults are called exception traps.

Your AMX Target Parameter File identifies which, if any, of the processor's exception traps are to be serviced by AMX. You can alter the Target Parameter File to suit your needs as described in the processor specific AMX Target Guide.

All of the exception traps serviced by AMX are considered fatal. The processor saves information which is dependent upon the processor type and the particular exception trap. The processor then vectors to the AMX handler for the particular fault.

All AMX fatal exception trap handlers operate as follows. AMX creates a processor dependent stack frame in which it identifies the type and location of the fault and the processor register contents at the time of the fault. Details are provided in each AMX Target Guide. AMX then takes its fatal exit at *`cjksfatal`* (see Chapter 14.1).

In general, there is little that can be done to recover from these fatal exception traps. Your Fatal Exit Procedure might be able to externally display the information which is provided for diagnostic purposes.

4.9 Vector Table Initialization

The manner in which the processor vectors to an Interrupt Service Procedure (ISP) is processor dependent. Most processors use a dispatch table which, in AMX nomenclature, is called a Vector Table. AMX maintains its own Vector Table if none is provided by the processor.

The Vector Table may be located in RAM or ROM as dictated by your hardware configuration. If the Vector Table is in RAM, it can be further characterized as alterable or not. All of these characteristics are defined in your AMX Target Parameter File as described in the processor specific AMX Target Guide.

Alterable Vector Table

The Vector Table must be initialized to provide dispatch access to all ISPs. If the Vector Table is in RAM and is declared alterable, AMX services can be used to initialize the Vector Table. Your application must initialize the Vector Table entry for each ISP root defined in your AMX Target Parameter File. You can use the AMX target specific procedure *`cjksivtwr`* or *`cjksidtwr`* during the AMX launch to install any ISP (including an NMI ISP or other nonconforming ISPs) into the Vector Table entry of your choice.

For most processors, the Vector Table must also provide access to all exception trap handlers. If the Vector Table is in RAM and is declared alterable, AMX will automatically install its own exception trap handlers for the exceptions which AMX has been configured to handle.

Unalterable Vector Table

Special consideration is required if the Vector Table is to be in ROM or is to be unalterable by AMX in the target AMX system.

The Vector Table must be initialized when the ROM image is created. Pointers to all ISPs (including conforming ISP roots, nonconforming ISPs and the NMI ISP) must be installed in the ROM image.

In addition, AMX requires that the entries for all exception traps for which it is responsible be initialized. The processor dependent list of AMX exception trap handlers can be found in the processor specific AMX Target Guide.

If the Vector Table is in RAM but has been declared unalterable, the RAM copy of the Vector Table must be initialized (as for a ROM image) before AMX is launched.

This page left blank intentionally.

5. AMX Timing Control

5.1 Introduction to Timing Facilities

Most real-time systems are characterized by the need to provide precise control over the timing of activities. A hardware clock provides the timing source; AMX provides the control over its use.

The unit of time in an AMX system is the system tick which is a fixed interval derived from the hardware clock. The system tick interval is user selectable. Typically, it is set at 10 ms or 100 ms. The system tick interval is chosen to provide the minimum resolution required in a particular application without inflicting unnecessary timing overhead.

Task Delays and Timeouts

A task can suspend itself for a specific interval. A task can also wait for an event which must occur within a specific interval. If the event fails to occur within the interval, the task resumes execution with a timeout indication.

Interval Timers

Application interval timers are the most general form of timer provided by AMX. Once such a timer has been created, it can be started, interrogated and stopped by any task or Interrupt Service Procedure. When a timer is created, an application dependent Timer Procedure must be provided. Whenever the timer expires, AMX executes this Timer Procedure passing it a parameter which was specified when the timer was created.

Application timers can also be periodic. The timer period is specified when the timer is created. AMX calls the corresponding Timer Procedure periodically at the defined interval.

The following AMX procedures provide interval timer services:

`cjtmbuild`	Create an interval timer (using a definition structure)
`cjtmconvert`	Convert milliseconds to system ticks
`cjtmcreate`	Create an interval timer (using inline parameters)
`cjtmdelete`	Delete an interval timer
`cjtmread`	Read an interval timer
`cjtmtick`	Read absolute AMX tick counter
`cjtmwrite`	Start/stop an interval timer

Calendar Clock

The AMX Time/Date Manager provides Y2K compliant time of day calendar support if required. The AMX calendar clock includes second, minute, hour, day, month, year and day of the week. AMX services are provided to set and read the calendar clock. A formatting procedure is also provided to translate the calendar time and date from the internal format in which it is maintained by AMX into an ASCII string in several of the most popular formats.

An application procedure can be tied to the calendar clock and called at one second intervals to permit simple time of day event scheduling.

The following AMX procedures provide calendar clock services:

cjtdfmt	Format calendar time/date as an ASCII string
cjtdget	Read calendar time/date
cjtdset	Set calendar time/date

Time Slicing

The AMX Timer Manager can provide time slicing if required. Time slicing is the process whereby AMX can force tasks which have the same execution priority to share the use of the processor on a round robin basis. The processing interval allocated to a task is called its time slice. The time slice for each task can be uniquely defined permitting fine tuning of the AMX time slice facility to meet each application's particular needs.

The following AMX procedures provide time slicing services:

cjtmslice	Change a task's time slice interval
cjtmtsopt	Enable or disable time slicing

5.2 AMX Clock Handler and Kernel Task

AMX includes a conforming clock ISP root, a Clock Handler and a Kernel Task to provide timing facilities. Whenever a clock interrupt occurs, the clock ISP root calls your application clock Interrupt Handler to dismiss the hardware clock interrupt. The ISP root then calls the AMX Clock Handler to trigger the AMX Kernel Task if required. The Kernel Task is triggered at the defined system tick interval if, and only if, there is any outstanding timing activity required in the system. In this case, the interrupted task is suspended and the AMX Kernel Task begins execution.

The AMX Kernel Task monitors all tasks which are in a timed wait state. The timer used by AMX for task waits is maintained privately by AMX for each task. If the timer expires, the Kernel Task removes the task from the wait state. The task is allowed to resume execution when the Kernel Task ends with an indication that a timeout occurred.

The AMX Kernel Task also services all expiring application interval timers. Whenever an interval timer expires, the corresponding Timer Procedure is executed. This procedure can invoke a subset of the AMX services to trigger tasks, send messages to mailboxes or message exchanges, signal events or wake tasks. If the timer is defined to be periodic, the AMX Kernel Task automatically restarts it with its predefined period.

Once all expiring task timers and application interval timers have been serviced, the AMX Kernel Task ends execution.

To install the AMX clock ISP, you edit the AMX Target Parameter File to include the name of your clock Interrupt Handler. The AMX Configuration Generator is then used to convert your Target Parameter File to an AMX Target Configuration Module with its embedded AMX clock ISP root.

Your clock Interrupt Handler can be coded in either C or assembler. For efficiency, assembler is recommended since most handlers require only one or two machine instructions to dismiss the clock interrupt request.

You must also start your hardware clock at the correct frequency when AMX is launched. You can do this in a Restart Procedure or in some task which is triggered at launch time.

Note

AMX is delivered to you with clock Interrupt Handlers ready for use with one or more of the timing devices commonly used with the target processor.

5.3 Interval Timers and Timer Procedures

AMX supports any number of application interval timers in a system. The maximum number in a system is defined in your System Configuration Module (see Chapter 15.4).

A timer must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create timers. It is recommended that only Restart Procedures and tasks be used to create timers.

AMX procedure *cjtmbuild* or *cjtmcreate* is used to create a timer. AMX allocates a timer and returns a timer id to the caller. The timer id is a handle which uniquely identifies the particular timer allocated for use by the application. It is the responsibility of the application to keep track of the timer id for future reference to the timer.

When a timer is created, you can provide a unique 4-character tag to identify the timer. The tag can be used subsequently in a call to *cjksfind* to find the timer id allocated by the Timer Manager to the particular timer.

When a timer is created, the caller must also specify the following parameters: the timer period, a pointer to an application Timer Procedure and an optional 32-bit application dependent parameter.

The timer period determines if the timer is periodic. If the timer period is zero, the timer is a one-shot timer. Whenever a one-shot timer is started, it runs until it expires at which time it remains idle until started again.

If the timer period is non-zero, the timer is periodic. The period specifies the timer's period measured in AMX system ticks.

Whenever a timer expires, the AMX Kernel Task executes the Timer Procedure which was provided when the timer was created. The Timer Procedure receives the timer id and the predefined 32-bit application parameter as parameters.

When a timer is created, AMX sets the timer idle. The timer remains idle until it is started by a Restart Procedure, task, ISP or Timer Procedure. Timers are started by calling AMX procedure *cjtmwrite* to write the initial timer interval into the timer.

Timer intervals are measured in multiples of system ticks. For convenience, the AMX procedure *cjtmconvert* can be used to convert a period specified in milliseconds to the corresponding number of system ticks.

Timers are down-counters. When a timer expires, the AMX Kernel Task calls the timer's Timer Procedure. One-shot timers remain expired unless they are restarted by their Timer Procedure. Periodic timers are automatically restarted by the AMX Kernel Task with their predefined timer period before the timer's Timer Procedure has been executed.

When an interval timer is no longer needed, it can be deleted with a call to *cjtmdelete*.

The AMX service procedures which can be called from a Timer Procedure include the following:

cjtktrigger	Start (trigger) a task
cjtkwake	Wake a task which is waiting
cjmbsend	Send a message to a mailbox Must not wait for acknowledgement
cjmxsend	Send a message to a message exchange Must not wait for acknowledgement
cjtmcreate	Create an interval timer
cjtmdelete	Delete an interval timer
cjtmread	Read an interval timer
cjtmwrite	Start/stop an interval timer
cjtmconvert	Convert milliseconds to system ticks
cjsmsignal	Signal to a semaphore
cjevsignal	Signal one or more events in an event group
cjbmget	Get a buffer from a specific buffer pool
cjbmfree	Free a buffer
	Circular List Manager services Linked List Manager services

The following conditions exist when the Timer Procedure is called by the AMX Kernel Task:

Interrupts are enabled.
All registers are free for use.
The AMX Kernel Stack is in use.

Timer Procedures are written as C procedures with formal parameters.

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

void CJ_CCPP truser(
CJ_ID       timerid,                     /* Timer id               */
struct userblock *userp)                 /* Pointer to user block  */
{
   :
   Do timer expiry processing
   :
   }
```

Timerid is the timer id assigned by AMX to the interval timer when it was created. *Userp* is the 32-bit application parameter provided when the timer was created. In this example, it is assumed that *userp* is a pointer to an application structure of type *userblock*.

The Timer Procedure must execute with the interrupt facility enabled. If interrupts must be temporarily disabled, they must be enabled prior to returning to the AMX Kernel Task. A Timer Procedure must not issue any AMX directives which would in any way force the Kernel Task to wait for any reason.

Note

Timer Procedures must only use AMX services which are marked in Chapter 18 as
■ Timer Procedure

Application Timer Procedures use the AMX Kernel Stack.

In addition to the minimum stack size required for the AMX Kernel Stack, you must allocate sufficient stack to satisfy the worst case requirements of all application Timer Procedures.

5.4 Task Time Slicing

AMX provides task time slicing as an option. The AMX system must be configured to include a clock if time slicing is to be possible. Time slice intervals are then specified as multiples of the AMX system tick.

Time slicing is normally disabled. It is enabled with a call to AMX procedure *cjtmtsopt(CJ_YES)*. It can be disabled again with a subsequent call to *cjtmtsopt(CJ_NO)*.

If your AMX System Configuration Module indicates that time slicing is required, AMX will automatically enable time slicing by calling *cjtmtsopt* at launch time after completing its own initialization, but prior to executing your application Restart Procedures.

Time slicing is a feature which coexists with the normal AMX preemptive priority-based task scheduling. Whether or not a task is time-sliced depends on two things: the task's execution priority and the task's time slice interval. Both of these parameters are first defined when a task is created.

If a task's time slice interval is zero, the task will not be time sliced. A non-zero time slice interval specifies the number of AMX system ticks which will be given to the task before the task is forced to relinquish the processor to another time sliced task.

A task is time sliced with all other tasks having the same execution priority and a non-zero time slice interval. A set of time sliced tasks are normally created to share a particular execution priority. Tasks which are not time sliced (their time slice interval is zero) are usually assigned to different execution priority levels. More than one set of time sliced tasks can be created, each set existing at a different priority level. Other tasks can reside at priorities between the time sliced sets.

The order in which tasks at a particular shared priority level are given their time slice by AMX is determined by the chronological order in which the tasks were created.

Figure 5.4-1 illustrates the allocation of processing time to two tasks, B and C. Task B was created first with a time slice interval of 100 AMX system ticks. Task C was created later with a time slice interval of 50 AMX system ticks. At time *t1*, tasks B and C were both triggered by a higher priority task A which then ended.

Since task B was created first, it executes first. Thereafter, in the absence of any other higher priority task activity, the processor is shared by tasks B and C as illustrated.

Task C begins to execute at time *t2* when task B's first 100 tick time slice expires. At time *t3*, task B ends execution. The processor is immediately given to task C which continues to execute without further interruption by task B.

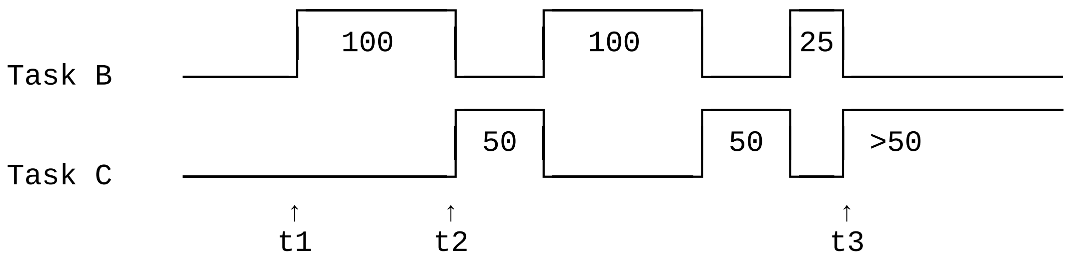


Figure 5.4-1 Simple Time Slicing

Figure 5.4-2 illustrates the effects to be expected if, while tasks B and C are being time sliced, higher priority task A is invoked and executes to completion within 125 system ticks.

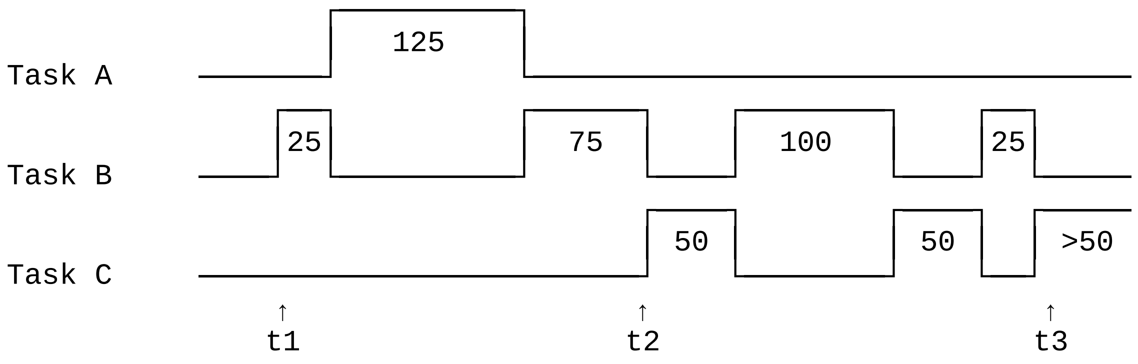


Figure 5.4-2 Interrupted Time Slicing

Time slicing can be completely disabled at any time with a call to AMX procedure *cjtmtsopt(CJ_NO)*. Procedure *cjtmtsopt(CJ_YES)* can then be used to enable time slicing.

Task time slice intervals can be dynamically adjusted using AMX procedure *cjtmslice* to fine tune the shared use of the processor based on observed effects.

The starting and ending of time sliced tasks are not synchronized to the AMX clock. However, switching between time sliced tasks will only occur at AMX system ticks because the switch is initiated by the AMX Kernel Task.

It should be noted that if higher priority tasks frequently interrupt a time sliced task, the exact processing time allocated to the task will not exactly equal *n*t* where *n* is the task's time slice interval and *t* is the AMX system tick period. The task will be forced to relinquish the processor after the *n*'th system tick which occurs while the task is executing. However, because of higher priority task activity, the task did not get the use of the processor for the entirety of each of its *n* system ticks.

Finally, if a task is time sliced and all other tasks at the same execution priority are idle or blocked, a time slicing overhead penalty will still exist. The AMX Kernel Task will continue to interrupt the task at its time slice interval, conclude that task switching is unnecessary and resume execution of the interrupted task.

Suggestion

Do not use time slicing unless its use is warranted. Better use of processor time results if tasks are allowed to run to completion.

Misuse is abuse!

5.5 Time/Date Manager

Most real-time systems require the maintenance of a calendar clock. The AMX Time/Date Manager provides this facility.

The Y2K compliant calendar clock maintained by the Time/Date Manager includes second, minute, hour, day, month and year. Leap year is accounted for. The day of the week (Mon to Sun) is also maintained.

An application Scheduling Procedure can be connected to the calendar clock to provide activity scheduling based on the time of day.

The Time/Date Manager includes a set of service procedures for use by application tasks, Timer Procedures, Interrupt Service Procedures and Restart Procedures. These include:

`cjtdget`	Get Time and Date
`cjtdset`	Set Time and Date
`cjtdfmt`	Format Time and Date as an ASCII string

Several conflicts always arise with the maintenance of time and date. These conflicts arise when:

1) the clock is trying to update the time and date while
2) a task is trying to read the time and date and
3) another task is trying to set the time and date.

The conflicts are even more pronounced if separate calls are required to get both time and date. The Time/Date Manager resolves these conflicts.

Operation

The Time/Date Manager includes two components: an AMX Restart Procedure and a set of service procedures.

If your AMX System Configuration Module enables the Time/Date option, AMX automatically calls the Time/Date Restart Procedure during the launch prior to executing any application Restart Procedure. The calendar clock is set to 00:00:00 Friday Jan 01/1993 (the distribution date of the Time/Date Manager).

The Time/Date Manager creates a periodic interval timer and starts it with a one second period. At one second intervals thereafter, the AMX Kernel Task executes the Time/Date Timer Procedure. The current time and date are updated by one second.

The Time/Date Manager employs an interlock mechanism to ensure that any race between a task trying to set a new time and date and the Time/Date Timer Procedure trying to update time and date is resolved. The task's new time and date have precedence.

This interlock mechanism also assures that any task trying to read the current time and date does not get a partially updated (and hence erroneous) time and date.

Once the time and date have been updated, the Time/Date Timer Procedure calls an application Scheduling Procedure if one has been provided. This procedure, called once a second, is thus tied to the calendar clock and can be used to initiate activities which must be time of day driven. A detailed description of this facility is provided later in this chapter.

The Time/Date Timer Procedure ends once the application Scheduling Procedure (if any) has been executed.

Time/Date Structure

The Time/Date Manager provides time and date in the following form. Structure *cjxtd* is defined in the AMX header file *CJZZZSD.H* as follows:

```
/* AMX Time/Date Structure                                            */

struct cjxtd {

   unsigned char xtdsec;            /* seconds (0-59)               */
   unsigned char xtdmin;            /* minutes (0-59)               */
   unsigned char xtdhr;             /* hours   (0-23)               */
   unsigned char xtdday;            /* day     (1-31)               */
   unsigned char xtdmonth;          /* month   (1-12)               */
   unsigned char xtdyear;           /* year    (0-99)               */
   unsigned char xtdow;             /* day of week (Mon=1 to Sun=7) */
   unsigned char xtdcen;            /* 0 if time/date is incorrect  */
                                    /* century if time/date is      */
                                    /* correct                      */
   CJ_ID    xtdid;                  /* Time/Date timer id           */
   };
```

Time/Date Validity

The century is used as follows. At startup, the Time/Date Restart Procedure resets the century to 0 to indicate that the initial default time and date are incorrect. Note that the initial time and date are valid; they are just not correct. At some later time, an application program can issue a call to the Time/Date service procedure `cjtdset` to set a correct time and date. The century specified as a parameter in that call should be set indicating that the time and date parameters are correct.

The century can be reset to 0 by an application program with a `cjtdset` call to indicate that the time and date are incorrect.

When setting the time and date, the day of the week does not have to be provided as long as the year is beyond 1800. The Time/Date Manager will figure it out and set it accordingly.

Time/Date Scheduling

Many real-time systems require that certain activities be performed at specific times of day. For instance, it may be required that every day at 8:00 a.m. a 24 hour report be generated. Or maybe every 1/2 hour, measured from the hour, the system must make a measurement and display the result. The Time/Date Manager provides the mechanism necessary to implement such features.

At one second intervals the Time/Date Timer Procedure updates the time and date and then calls an application Scheduling Procedure. The name of this procedure must be provided in your System Configuration Module (see Chapter 15.8).

Your Scheduling Procedure is called with a pointer to a Time/Date structure as a parameter. The structure specifies the time and date at the instant the procedure is called.

The Scheduling Procedure is application dependent. It executes as part of the Time/Date Timer Procedure. It must therefore not be compute or I/O bound.

In general, the procedure checks the time (and date if necessary) and determines if some application dependent action must be initiated at that instant. If action is required, the procedure either performs the action directly or requests AMX to start some other task which is responsible for taking the required action.

You must be sure to allocate sufficient stack to the AMX Kernel Task to accommodate the needs of the Scheduling Procedure.

The following conditions exist when your Time/Date Scheduling Procedure is called by the Time/Date Manager

Interrupts are enabled.
All registers are free for use.
The AMX Kernel Stack is in use.

The Scheduling Procedure is written as a C procedure as follows:

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

void CJ_CCPP tdshed(struct cjxtd *tdp)
{

   :
   Perform required tests and initiate
   actions if it is time for them
   :
   }
```

Time/Date ASCII Formats

The Time/Date Manager procedure `cjtdfmt` can be used to format time and date into an ASCII character string in any of several popular formats. The time and date is presented to `cjtdfmt` in the standard AMX time/date structure. The ASCII string is returned in a character buffer provided by the caller. See Chapter 18 for the `cjtdfmt` calling sequence.

Time can be formatted as either of:

```
HH:MMb
HH:MM:SSb
```

where:	*HH*	= hours	(00 to 23)	
	MM	= minutes	(00 to 59)	
	SS	= seconds	(00 to 59)	
	b	= space	time/date are correct	(century <> 0)
	b	= #	time/date are incorrect	(century = 0)

The date can be formatted as any of:

```
DDD MMM dd/ccyyb
DDD mm/dd/ccyyb
DDD dd MMM/ccyyb
DDD dd/mm/ccyyb
DDD ccyy/mm/ddb
```

where:	*DDD*	= day of week	(Mon to Sun)
	MMM	= month	(Jan to Dec)
	mm	= month	(01 to 12)
	dd	= day	(01 to 31)
	cc	= century	(19, 20, ...)
	yy	= year	(00 to 99)
	b	= space	

The 4 character day-of-week (*DDDb*) and/or the two character century (*cc*) can be omitted.

The format of time and/or date is specified by a format specification parameter, a single byte presented to procedure `cjtdfmt`.

Figure 5.5-1 describes the format specification byte and the effect of each bit in it on the formatting of the time and date.

7	6	5	4	3	2	1	0
TD1	*TD0*	*C*	*M*	*S*	*W*	*D1*	*D0*

TD1	*TD0*	Time/Date selection	
0	0	time followed by date	23:59:59 Sun Jan 31/93
0	1	time only	23:59:59
1	0	date only	Sun Jan 31/93
1	1	date followed by time	Sun Jan 31/93 23:59:59

S = 0	display seconds	23:59:59
S = 1	suppress seconds	23:59

W = 0	suppress day of week	23:59:59 Jan 31/93
W = 1	display day of week	23:59:59 Sun Jan 31/93

M	*D1*	*D0*	Date format	
0	0	0	American alphanumeric form	Jan 31/93
0	0	1	American numeric form	01/31/93
0	1	0	European alphanumeric form	31 Jan/93
0	1	1	European numeric form	31/01/93
1	0	0	Metric form	93/01/31

C = 0	suppress century	Jan 31/93
C = 1	include century	Jan 31/1993

Figure 5.5-1 Time/Date Format Specification Parameter

This page left blank intentionally.

6. AMX Semaphore Manager

6.1 Introduction

E.W. Dijkstra introduced two primitive operations to resolve two seemingly unrelated problems: mutually exclusive access by tasks to critical resources and the synchronization of asynchronously occurring activities.

The abstract primitives, called P and V operators, operate on a variable called a semaphore. Many variations of these P and V operators have been implemented since their first introduction. The AMX Semaphore Manager provides two variations: a counting semaphore which has been enhanced to provide priority queuing and automatic timeout, and a resource semaphore in which resource ownership is tied to a specific task.

A **counting semaphore** is a semaphore with an associated counter which is incremented by the P operator (signal) and decremented by the V operator (wait). The item controlled by the semaphore is free (available) when the counter is greater than 0. The upper limit for a counting semaphore is fixed at 16383. A counting semaphore is best used to signal events without losing count of the events.

A **bounded semaphore** is a counting semaphore with a fixed upper limit between 1 and 16383. It can be used to control access to a specific number of items. The number of items controlled by the semaphore is determined by the maximum value the counter is allowed to achieve.

A **binary semaphore** is a bounded semaphore with an upper limit of one. It can be used to provide mutually exclusive access to a single item or to signal one particular event.

A **basic resource semaphore** is a binary semaphore which can only be used for resource ownership control. It differs from a pure binary semaphore in one significant feature: resource ownership is tied to a specific task. No other task except the task owning the resource is allowed to signal the release of the resource.

A **priority inheritance resource semaphore** is a variation of a basic resource semaphore which can be used to avoid the ever present danger of task priority inversion. This well documented phenomenon will occur if a high priority task is forced to wait for a long time for a resource owned by a low priority task which is prevented from running by intervening medium priority tasks. The inversion is prohibited by the Semaphore Manager which hoists the task owning the resource to a priority immediately above that of the high priority task demanding access to the resource. Once the resource is released, ownership is granted to the high priority task which resumes execution immediately. Eventually, the original owner is allowed to resume execution at its original priority.

The semaphores provided by the Semaphore Manager have been enhanced to provide priority queuing and automatic timeout. Tasks which wait on semaphores can specify the priority at which they wish to wait for the resource or event controlled by the semaphore. Tasks can also specify the maximum interval which they are prepared to wait.

The AMX Semaphore Manager provides the following semaphore management services:

`cjrmbuild`	Create a resource semaphore (using a definition structure)
`cjrmcreate`	Create a basic resource semaphore (using inline parameters)
`cjrmcreatex`	Create a basic or priority inheritance resource semaphore
`cjrmdelete`	Delete a resource semaphore
`cjrmfree`	Free a resource semaphore (unconditional)
`cjrmrls`	Release a resource semaphore (nested)
`cjrmrsv`	Reserve a resource semaphore (optional timeout)
`cjrmstatus`	Read status of a resource semaphore
`cjsmbuild`	Create a counting/bounded semaphore (using a definition structure)
`cjsmcreate`	Create a counting/bounded semaphore (using inline parameters)
`cjsmdelete`	Delete a counting/bounded semaphore
`cjsmsignal`	Signal to a counting/bounded semaphore
`cjsmstatus`	Read status of a counting/bounded semaphore
`cjsmwait`	Wait on a counting/bounded semaphore (optional timeout)

Your use of the Semaphore Manager is optional. If you intend to use it, you must indicate so in your System Configuration Module. You must also provide a hardware clock and include the AMX timing facilities.

Semaphores can be predefined in your System Configuration Module which is processed by the Semaphore Manager at startup. Semaphores which are predefined are automatically created by the Semaphore Manager. The semaphore id assigned to each predefined semaphore is stored in a variable which you must provide for that purpose.

6.2 Semaphore Use

The Semaphore Manager supports any number of semaphores. The maximum number of semaphores in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual semaphores that can be created in your application.

A semaphore must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create semaphores. It is recommended that only Restart Procedures and tasks be used to create semaphores.

A counting semaphore is created with a call to procedure *cjsmbuild* or *cjsmcreate*. A resource semaphore is created with a call to procedure *cjrmbuild*, *cjrmcreate* or *cjrmcreatex*. The Semaphore Manager allocates a semaphore and returns a semaphore id to the caller. The semaphore id is a handle which uniquely identifies the semaphore. It is the responsibility of the application to keep track of the semaphore id for future reference to the semaphore.

When a semaphore is created, you can provide a unique 4-character tag to identify the semaphore. The tag can be used subsequently in a call to *cjksfind* to find the semaphore id allocated by the Semaphore Manager to the particular semaphore.

At any time, the status of a counting semaphore can be read using *cjsmstatus* to determine if an item is available or if tasks are waiting for an item.

At any time, the status of a resource semaphore can be read using *cjrmstatus* to determine if a resource is available or if tasks are waiting to use the resource.

When a semaphore is no longer needed, it can be deleted with a call to *cjrmdelete* or *cjsmdelete*. The Semaphore Manager will reject the attempt to delete the semaphore if any task is waiting for the use of the semaphore. When the Semaphore Manager deletes a semaphore, it marks the semaphore as invalid such that any subsequent reference to the semaphore will be rejected.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the semaphore just as you go to delete it. Be aware that the deleted semaphore id may immediately be reused by AMX for some other purpose.

Counting, Bounded and Binary Semaphores

A **counting semaphore** is created with a call to *cjsmbuild* or *cjsmcreate* in which the initial value of the semaphore count must be specified. The initial value must lie in the range 0 to 16383. The upper limit for a counting semaphore's count is fixed at 16383.

A **bounded semaphore** is created with a call to *cjsmbuild* or *cjsmcreate* in which the upper limit of the semaphore count must be specified. The upper limit must lie in the range 1 to 16383. The bounded semaphore's initial count is set to 0.

A **binary semaphore** is created with a call to *cjsmbuild* or *cjsmcreate* in which the semaphore is declared to be binary. The binary semaphore's initial count is set to 0 and its upper limit is set to 1. There is no difference between a binary semaphore and a bounded semaphore with an upper limit of 1. Use a binary semaphore for mutual exclusion.

If a semaphore is initialized with a semaphore value of *n*, it can be used to control access to *n* items of a particular type. For instance, a bounded semaphore could be used to control access to three printers in a system. The bounded semaphore would have to be signaled three times to set its count to three to indicate that the three printers are available. When used in this fashion, the Semaphore Manager assures that no more than three tasks can own a printer at any one time. However, the Semaphore Manager does not provide any guidance as to which of the three available printers a task can use.

Access to the item controlled by a counting or bounded semaphore is acquired with a call to procedure *cjsmwait*. Only tasks are allowed to call this procedure and wait for the item to become available. When requesting use of an item, the task specifies the interval, measured in system ticks, which it is willing to wait for the item if it is unavailable at the time of the call. The task also specifies the priority at which it is prepared to wait, zero being the highest priority.

If the item controlled by the semaphore is available, the Semaphore Manager gives it to the calling task immediately. If the item is unavailable, the Semaphore Manager will add the task to the semaphore's wait queue at the priority it said it was willing to wait. Thus tasks which require high priority access to the item can preempt lower priority waiting tasks in the wait queue.

When the task is finished using the item, it signals its release by calling procedure *cjsmsignal*. The Semaphore Manager releases the item and checks the semaphore's wait queue. If any tasks are waiting on the queue, the item is immediately given to the task at the head of the wait queue. If that task is of higher priority than the task which is releasing the item, a task switch will occur giving the higher priority task an immediate opportunity to use the item. If the task being granted use of the item is of lower priority than the task which is releasing it, the new owner will have to wait until the currently executing task relinquishes control of the processor.

If the semaphore does not become available within the timeout interval specified by the task, the task will be removed from the semaphore wait queue and will resume execution with a timeout indication.

Tasks, ISPs or Timer Procedures which need to use an item but which cannot wait for the item to be free can still call *cjsmwait*. However, they must specify a timeout value of less than zero so that they will not wait on the semaphore queue. If the item cannot be immediately granted to the caller, *cjsmwait* returns an error indication.

Basic Resource Semaphore

The AMX basic resource semaphore provides the simplest mechanism for controlling access to critical resources. Resources may include disk files, I/O devices, database components, regions of memory, specific words of memory or any other entity which is considered to be a resource.

An application task requests ownership of a resource with a call to the Semaphore Manager. If the resource is available, the task is granted immediate ownership of it. If the resource is unavailable, the task is inserted on a list of tasks waiting for the resource at a wait priority specified by the task. The task can optionally specify the maximum time interval it is prepared to wait for access. When the task which currently owns the resource releases it with a call to the Semaphore Manager, the resource will be given to the task with the highest waiting priority.

Although a basic resource semaphore uses a binary semaphore for controlling access to the resource, it differs from the general bounded (binary) semaphore provided by the AMX Semaphore Manager in one significant feature. Resource ownership is tied to a specific task. Only the task owning such a resource is permitted to signal its release. The Semaphore Manager does not permit more than one task to share ownership of such a resource.

A basic resource semaphore is created by a call to *cjrmbuild* or *cjrmcreate*. The Semaphore Manager creates a resource semaphore and automatically gives it an initial value of one indicating that the resource is free.

A resource is reserved by calling procedure *cjrmrsv* using the semaphore id allocated to the particular resource semaphore when it was created. Only tasks can reserve a resource controlled by a resource semaphore. The task which owns the resource can reserve it again and again, resulting in a nested reservation.

When requesting use of a resource, the task specifies the interval, measured in system ticks, which it is willing to wait for the resource if the resource is unavailable at the time of the call. The task also specifies the priority at which it is prepared to wait, zero being the highest priority.

The resource is released with a call to *cjrmrls*. The task which owns the resource must release the resource once for each nested reserve that it has made. The resource becomes free when the nesting count reaches zero. Alternatively, the resource can be released unconditionally with a call to *cjrmfree*.

If the resource is in use when *cjrmrsv* is called, the task will be added to the semaphore's wait queue forcing the task to wait at the priority specified by the task in its call to reserve the resource. When the current owner of the resource releases it, the Semaphore Manager gives the resource to the task (if any) waiting at the head of the resource semaphore queue. Hence, a task must wait until all other tasks ahead of it in the queue use and release the resource.

If the task is of higher priority than the task which is releasing the resource, a task switch will occur giving the higher priority task an immediate opportunity to use the resource. If the task being granted use of the resource is of lower priority than the task which is releasing it, the new owner will have to wait until the currently executing task relinquishes control of the processor.

If the resource does not become available within the timeout interval specified by the task, the task will be removed from the resource semaphore wait queue and will resume execution with a timeout indication.

Tasks which need to use a resource but which cannot wait for the resource to be free can still call *cjrmrsv*. However, they must specify a timeout value of less than zero so that they will not wait on the resource semaphore queue. If the resource cannot be immediately granted to the caller, *cjrmrsv* returns an error indication.

Priority Inheritance Resource Semaphore

The priority inheritance resource semaphore is a basic resource semaphore which is enhanced to use priority inheritance to prevent task priority inversions. Such a semaphore is created by a call to *cjrmbuild* or *cjrmcreatex* in which the semaphore is declared to support priority inheritance.

The semaphore operates like a basic resource semaphore with one significant difference: a high priority task is never permitted to block (be suspended), waiting for the use of such a resource. Instead, the task owning the resource is hoisted to a priority immediately above that of the task requiring the resource so that it can continue to execute until it releases the resource. Ownership is immediately granted to the high priority task and the prior owner is forced to wait until it can resume execution at its original priority.

A task which owns a priority inheritance resource is not permitted to block for any reason. The task will receive an error code from any AMX procedure which would have to suspend the task. For example, the task cannot delay, wait for a wake request or wait on a buffer pool, an event group, a mailbox or a message exchange. The task cannot send a message and wait for acknowledgement of its receipt.

A task which owns a priority inheritance resource can wait for another resource of the same type. However, it cannot wait for a counting, bounded or binary semaphore or for a basic resource semaphore.

The Semaphore Manager uses the AMX task suspension and priority adjustment services to control the execution of tasks competing for a priority inheritance resource. Consequently, your application must avoid calls to *cjtksuspend*, *cjtkpriority* or *cjtkpradjust* to suspend any task or alter the priority of any task which manipulates priority inheritance resources.

A task can own more than one priority inheritance resource. It is recommended that, if possible, resources should be released in the opposite order to which they were reserved. To avoid deadlock, all tasks which compete for such resources should reserve the resources in the same order. It is also recommended that each task execute at its own distinct priority level.

Tasks which use priority inheritance resources should not be time sliced. Once a time sliced task has been hoisted in priority to resolve a request for a resource that it owns, the task will not be sliced out until it releases the resource. At that time it will be suspended as though its time slice had just expired. The task will eventually resume execution at its original priority with a full time slice when its next turn to execute at that priority arises.

6.3 Semaphore Applications

Mutual Exclusion

Assume that three tasks, A, B and C, require shared access to a common data structure being used to control some process. Access to the data structure must be mutually exclusive so that one task cannot be modifying the data in the structure while another task is accessing it.

A binary semaphore is required because there is only one data structure to manage. The initial count must be one to indicate that the data structure is free for use. A binary semaphore is created in a Restart Procedure. The semaphore count is set to one by signaling the semaphore. One task can own the resource; two tasks can be waiting. Note that the Semaphore Manager does not guarantee that only tasks A, B and C can access the data structure.

In this example, Task A waits indefinitely (timeout value = 0) at priority level 20 for access to the data variable. Task B waits indefinitely at priority level 10. Task C only waits for 100 system ticks at priority 5 before it assumes that it cannot have access.

This example is illustrated on the next page. The manner in which Tasks A, B and C are created and started is beyond the scope of this example.

Parameters 1 and 2 in the data structure are modified by the tasks to illustrate that their alteration by one task is guaranteed by the semaphore to be completed before access by either of the other two tasks is allowed.

For simplicity, the Restart Procedure does not check the error code returned by `cjsmcreate` or `cjsmsignal`. In practice, error codes should never be ignored.

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

static CJ_ID daccess;                   /* Data access semaphore id */

static struct {
   int      dbpar1;                     /* Parameter 1               */
   int      dbpar2;                     /* Parameter 2               */
   :
   :
   } datavar;                           /* Data variable             */


void CJ_CCPP rruser(void)               /* Restart Procedure         */
{
   /* Create binary semaphore; set count to 1 to allow data access    */
   cjsmcreate(&daccess, "DACS", CJ_SMBINARY);
   cjsmsignal(daccess);
   }


void CJ_CCPP sttaskA(void)              /* Task A                    */
{
                              /* Wait for access; priority 20 forever */

   if (cjsmwait(daccess, 20, 0) == CJ_EROK) {
            datavar.dbpar1 = 1;         /* Set parameters            */
            datavar.dbpar2 = 1;
            cjsmsignal(daccess);        /* Allow access by others    */
            }
   }


void CJ_CCPP sttaskB(void)              /* Task B                    */
{
                              /* Wait for access; priority 10 forever */

   if (cjsmwait(daccess, 10, 0) == CJ_EROK) {
            datavar.dbpar1 = 2;         /* Set parameters            */
            datavar.dbpar2 = 2;
            cjsmsignal(daccess);        /* Allow access by others    */
            }
   }

void CJ_CCPP sttaskC(void)              /* Task C                    */
{
                              /* Wait for access; priority 5; 100 ticks*/

   if (cjsmwait(daccess, 5, 100) == CJ_EROK) {
            datavar.dbpar1 = 3;         /* Set parameters            */
            datavar.dbpar2 = 3;
            cjsmsignal(daccess);        /* Allow access by others    */
            }
   }
```

Task/Event Synchronization

A counting semaphore can be used to provide synchronization between a task waiting for an event and a task, ISP or Timer Procedure in which the event is detected. The following example assumes that a device ISP detects the event.

A task creates a counting semaphore with an initial count of zero.

The task then starts the device and waits on the semaphore. When the event of interest is detected by the ISP, it signals to the semaphore. The Semaphore Manager grants access to the waiting task which then resumes execution knowing that the event occurred.

For simplicity, the task and ISP do not check the error code returned by `cjsmcreate`, `cjsmdelete` or `cjsmsignal`. In practice, error codes should never be ignored.

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

static CJ_ID syncisp;                      /* ISP synchronization       */
                                           /* semaphore id              */

void CJ_CCPP sttask(void)
{
   int      status;


                                           /* Create counting semaphore*/
   cjsmcreate(&syncisp, "SISP", CJ_SMCOUNT(0));
   :
   Start device I/O
   :
                                           /* Wait 1 second for event  */
   status = cjsmwait(syncisp, 0, cjtmconvert(1000));

   if (status == CJ_EROK) {                /* OK?                       */
            :
            Event occurred within 1 second
            :
            }

   else if (status == CJ_WRTMOUT) {        /* Timeout?                  */
            :
            Event did not occur within 1 second
            :
            }

   else {                                  /* Fatal error               */
            :
            No such semaphore exists (who deleted it?)
            :
            }

   cjsmdelete(syncisp);                    /* Delete semaphore          */
   }
```

```
void CJ_CCPP isphand(void)                /* Interrupt Handler        */
{
   Dismiss interrupt request
   :
   cjsmsignal(syncisp);                   /* Signal to semaphore      */
   }
```

In this example, we have assumed that the AMX Target Configuration Module includes an ISP root for the device with procedure `isphand` declared as the Interrupt Handler.

Resource Nesting

Assume that two tasks, A and B, have to share a numeric coprocessor. Furthermore, these two tasks also must share a common library procedure `ncmath` which must use the coprocessor.

A resource semaphore must be used because ownership of the numeric coprocessor must be tied to one task at a time and the owner must be allowed to make nested resource reservation calls.

In the example, Task A waits indefinitely (timeout value = 0) at priority level 20 for the use of the numeric coprocessor. Once the task owns the coprocessor, it initializes it and calls `ncmath` to perform some mathematical operation using it. Task A then finishes using the coprocessor and releases it.

Task B does not use the coprocessor directly. It calls `ncmath` to perform a mathematical operation which requires the use of the coprocessor.

The math library procedure waits forever at priority 20 to reserve the coprocessor without knowing which task is calling. If `ncmath` is called by Task B while Task A owns the coprocessor, Task B will be suspended by the Semaphore Manager until Task A is finished with the coprocessor.

The manner in which Tasks A and B are created and started is beyond the scope of this example.

For simplicity, Tasks A and B do not check the error code returned by `ncmath` or by the Semaphore Manager. In practice, error codes should never be ignored.

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

static CJ_ID ncaccess;                    /* Coprocessor resource     */
                                          /* semaphore id             */

void CJ_CCPP rruser(void)
{
   cjrmcreate(&ncaccess, "MATH");         /* Create a basic           */
                                          /* resource semaphore       */
   }


CJ_ERRST CJ_CCPP ncmath(
CJ_ID       ncid)                         /* Semaphore id             */
{
   int      status;

   status = cjrmrsv(ncid, 20, 0);         /* Reserve coprocessor      */
   if (status == CJ_EROK) {
            :
            Use coprocessor and do math operation
            :
            status = cjrmrls(ncid);       /* Release coprocessor      */
            }
   return(status);
   }


void CJ_CCPP sttaskA(void)
{
   cjrmrsv(ncaccess, 20, 0);              /* Reserve coprocessor      */
   :
   Initialize coprocessor
   :
   ncmath(ncaccess);                      /* Common math operation    */
   cjrmrls(ncaccess);                     /* Release coprocessor      */
   :
   }


void CJ_CCPP sttaskB(void)
{
   :
   ncmath(ncaccess);                      /* Common math operation    */
   :
   }
```

6.4 Priority Inversion Avoidance

Priority Inheritance

The AMX Semaphore Manager provides a resource semaphore which uses priority inheritance to avoid a task priority inversion. This type of semaphore was introduced in Chapter 6.1 and described in Chapter 6.2. Although the elimination of potential priority inversions is a laudable goal, there are still penalties and pitfalls that must be addressed.

Raising and lowering task priorities are time consuming operations. Hence, an execution penalty is paid every time a high priority task requests use of a resource owned by a low priority task. The penalty is sufficiently minimal that the gain is usually worth the cost.

However, if several tasks compete in complex ways for a number of different priority inheritance resources, there can be a significant amount of thrashing as AMX raises and lowers task priorities to resolve the conflicting demands. For example, consider three tasks, A, B and C. Assume that low priority task C owns resources Ra and Rb. If task B preempts task C and requests resource Rb, task C will be hoisted above task B. If task A then preempts task C and requests resource Ra, task C will again be hoisted, this time above task A. If task C releases resource Ra before releasing resource Rb, Ra will be granted to task A and task A will run in preference to task C. When task A relinquishes use of the processor, task B will run and repeat its demand for resource Rb which task C still owns. Hence, task C will once more be moved to a priority just above task B. Note that four task priority changes were required to avoid the potential priority inversions.

In this example, task B will remain preempted until task C releases Ra or Rb. If task B specified a timeout in its request for the resource, the timeout interval may have expired before task B resumes. When task B resumes, if resource Rb is not available and a timeout occurred, task B will receive a warning that it timed out waiting for Rb. If resource Rb is available but a timeout occurred, task B will receive a warning that it missed its deadline. In this case, task B will not be granted the resource even though it was available when the timeout was finally observed.

To avoid an endless thrashing sequence with little chance for recovery, the Semaphore Manager aborts a task's request for a resource if the resource owner is hoisted many, many times and still does not release the resource. You can adjust the magic number to force an earlier abort by following the procedure described in Appendix D.2.

If you wish, you can reduce the AMX code footprint by eliminating AMX support for priority inheritance resources as described in Appendix D.2.

Priority Ceiling

One well defined method for avoiding priority inversion is to establish a priority ceiling for a resource. Any task wishing to use the resource must first raise its priority to the defined ceiling for that resource. The task can then use the resource and, when finished, restore its original priority.

The AMX Semaphore Manager does not provide a resource semaphore with a priority ceiling. However, any task can use AMX task priority adjustment services to manipulate its priority to implement a priority ceiling for a resource, with no semaphore needed.

7. AMX Event Manager

7.1 Introduction

The AMX Event Manager provides the most general form of event synchronization offered by AMX. The Event Manager provides a convenient mechanism for separating the tasks waiting for events from the tasks, Timer Procedures and Interrupt Service Procedures which can signal the event. The Event Manager also allows more than one task to simultaneously wait for a particular event. Tasks can wait for a particular combination of events or for any one of a set of events to occur.

The Event Manager provides a set of event flags which can be associated with specific events in your system. These event flags are provided in groups with 16 or 32 event flags per group. The number of event flags per group is dictated by the basic register width (integer size) of the target processor.

As an example, suppose that two tasks, A and B, must each wait until a motor turns on. Assume that an interrupt will occur when the motor is turned on.

It is assumed that Tasks A and B are completely independent; their processing is unrelated. One method of accomplishing the necessary synchronization is for both tasks to set software flags indicating that they are waiting for the motor and then call AMX procedure *cjtkwait* to wait for the motor to turn on. When the motor control ISP detects that the motor has started, it could check the wait flags for the two tasks and wake the tasks if necessary with calls to *cjtkwake*.

This solution does not provide good functional separation between processes. The motor control ISP must be aware that Tasks A and B are waiting for the motor to turn on. As you can imagine, this lack of functional separation is compounded when there are many types of events occurring in a system.

The Event Manager provides a more general solution to this problem. An event flag is defined to represent the state of the motor (off or on). A Restart Procedure initializes the event flag so that it matches the actual state of the motor. When Tasks A and B must wait for the motor, they do so by calling the Event Manager requesting to wait until the motor control event flag indicates that the motor is on. When the motor control ISP detects that the motor is on, it signals the event with a call to the Event Manager. The Event Manager wakes all tasks, including Tasks A and B, which are waiting for the motor to be on.

The AMX Event Manager provides the following event management services:

`cjevbuild`	Create an event group (using a definition structure)
`cjevcreate`	Create an event group (using inline parameters)
`cjevdelete`	Delete an event group
`cjevread`	Read current state of events in a group
`cjevsignal`	Signal one or more events in a group
`cjevstatus`	Get current status of an event group
`cjevwait`	Wait for all/any of a set of events in a group (optional timeout)
`cjevwaits`	Get state of events at completion of event wait

Your use of the Event Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must also provide a hardware clock and include the AMX timing facilities.

Event groups can be predefined in your System Configuration Module which is processed by the Event Manager at startup. Event groups which are predefined are automatically created by the Event Manager. The event group id assigned to each predefined event group is stored in a variable which you must provide for that purpose.

7.2 Event Synchronization

The AMX Event Manager supports any number of event groups in a system. Each event group includes 16 or 32 event flags. The maximum number of event groups in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual event groups that are available through the Event Manager in your application.

Each event in a group is represented by a boolean flag representing the state of the event. The event flags are represented in one 16 or 32-bit variable. It is recommended that the boolean states 0 and 1 be used as false and true indicators respectively. The zero state therefore represents no event. The one state indicates that the event has occurred.

Two types of event flags are supported: state driven or pulsed. State driven events are most useful for monitoring the state or condition of something in which the state is determined by one, and only one, piece of application code. The state of a motor is a good example.

Pulsed events are most useful to signal fleeting or rapidly changing conditions. Pulsed events free the designer from having to worry about who resets the event flag: the signaller or some signal waiter. An example might be a motor's speed rising above some upper limit.

State driven and pulsed events can coexist in the same event group. It is the event signaller which determines whether the events being signalled require a state change or a momentary change.

An event group must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create event groups. It is recommended that only Restart Procedures and tasks be used for this purpose.

Event Manager procedure *cjevbuild* or *cjevcreate* is used to create an event group. The Event Manager allocates an event group and returns a group id to the caller. The group id is a handle which uniquely identifies the particular event group allocated for use by the application. It is the responsibility of the application to keep track of the group id for future reference to the event group.

When an event group is created, you can provide a unique 4-character tag to identify the event group. The tag can be used subsequently in a call to *cjksfind* to find the event group id allocated by the Event Manager to the particular event group.

When an event group is created, the caller specifies the initial state that each of the event flags in the group is to assume. Hence, the process of creating a group automatically initializes all events in the group to a predefined state. The assignment of events to specific event flags in the event group is completely determined by the system designer.

Once an event group has been created, it can be used to synchronize tasks to any of the events which it represents. A task can wait for an event by calling procedure *cjevwait*. Only tasks can wait for events. If a task wishes to wait for more than one event, all of the events must be contained in the one event group specified by the task in its call to *cjevwait*.

When the task calls *cjevwait* to wait for an event, it specifies the group id of the event group containing the events of interest. It provides a 16 or 32-bit mask identifying which of the events are of interest and a 16 or 32-bit value indicating the particular state of interest for each of the selected events. The task also specifies one of two types of match criterion to be used for event detection. Tasks can wait for any one of the selected events in the group to achieve its specified state. Alternatively, the task can insist that all of the selected events must exactly match their specified state before an event match can be declared.

Finally, the task calling *cjevwait* must also specify the interval, measured in system ticks, which it is prepared to wait for the event match to occur. Upon return from *cjevwait*, the task receives status indicating whether an event match occurred within the expected time interval. Note that a task will not be forced to wait if the state of the event flags meets the task's match criterion at the time of the call.

Events are signalled by tasks, ISPs and Timer Procedures. The event is signalled with a call to procedure *cjevsignal*. The caller specifies the group id of the event group which contains the particular event. More than one event can be signalled in a single call to *cjevsignal*. The caller specifies a 16 or 32-bit mask identifying the particular event flags in the group and a 16 or 32-bit value which specifies the new state of each of these selected event flags.

The type of event is specified in the call to *cjevsignal*. Both state changes and pulsed events force the selected event flags to the state specified by the caller. If the events are pulsed, the selected event flags are then immediately reset to zero resulting in a momentary state change.

Whenever an event is signalled, the Event Manager determines if any tasks are waiting for any of the events in the particular group. If tasks are waiting on the group, the Event Manager checks the new state of the event flags to see if the event match criterion of any of the waiting tasks has been achieved. Whenever an event match is detected, the Event Manager wakes the task whose match criterion has been met. Providing that no tasks of higher priority are executing, the task will resume execution with an indication that the event combination for which it was waiting has occurred.

If a task needs to know the exact state of the event flags at the time its event match occurred or timed out, it can issue a call to the Event Manager procedure *cjevwaits*.

A task, ISP or Timer Procedure can determine the current state of the event flags at any instant with a call to *cjevread*. Alternatively, procedure *cjevstatus* can be used to fetch the event group status including the current event flags and a count of the tasks, if any, waiting on the group.

If an event group is no longer required, the group can be deleted with a call to the Event Manager procedure *cjevdelete*. The Event Manager will free the event group for reuse. The Event Manager will not allow you to delete an event group which has one or more tasks still waiting for events in the group.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the event group just as you go to delete it. Be aware that the event group id may immediately be reused by AMX for some other purpose.

7.3 Event Flag Application

The following example, coded in C, is provided to illustrate the use of the AMX Event Manager for event synchronization.

The example shows two tasks, A and B, which must be synchronized to the state of a motor. Task A must wait for the motor to be turned on. Task B must wait for the motor to be on and up to its maximum speed.

A 100 millisecond interval timer samples a motor control status register to determine the state of the motor. Whenever the motor state changes, the Timer Procedure signals the event. Note that the Timer Procedure signals both the on/off state and the motor speed simultaneously. It only signals changes in the motor state so that event synchronization overhead is minimized.

Note that the Timer Procedure receives its timer's id *`timerid`* and a parameter *`unused`*, neither of which is used by the procedure.

A Restart Procedure allocates an event group for motor control. Two of the event flags in the group are initialized to reflect the state of the motor at the time the system was started. The remaining event flags in the group are unused. A 100 millisecond periodic interval timer is created and started.

The creation and starting of Tasks A and B is outside the scope of this example. It is assumed that if the Restart Procedure is unable to create an event group for motor control, Tasks A and B will not be started. In our example, Tasks A and B assume that a valid event group id has been provided in variable *`motorgroup`*.

Note that bit 0 of the motor control status register determines if the motor is on or off. Bit 1 of the motor control status register determines if the motor is at maximum speed. For convenience, event flags 0 and 1 (bits 0 and 1 in the event group) are assigned to mirror the corresponding bits in the motor control status register.

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

static CJ_ID motorgroup;
static unsigned int motorstatus;

#define MOTORPORT 0x00700018L            /* Motor status port        */
#define MOTORON 0x01                     /* Motor on                 */
#define MOTORMAX 0x02                    /* Motor at maximum speed   */

#define MOTOREVT (MOTORON + MOTORMAX)



                                         /* Motor Timer Procedure    */
void CJ_CCPP tpmotor(CJ_ID timerid, void *unused)
{
   unsigned int status;

   status = cjcfin8(MOTORPORT) & MOTOREVT;

   /* If a change in motor status occurs                             */
   /*       Update motor status                                      */
   /*       Signal that changes have occurred                        */

   if (status != motorstatus) {
            motorstatus = status;
            cjevsignal(motorgroup, MOTOREVT, status, CJ_EVCONST);
            }
   }


void CJ_CCPP rrmotor(void)               /* Motor Restart Procedures */
{
   CJ_ID    timerid;

   motorstatus = cjcfin8(MOTORPORT) & MOTOREVT;

   /* If an event group is available                                 */
   /*       Set the initial event states to match the motor status   */
   /*       If a 100 ms periodic timer can be created                */
   /*                 Start it to go off at the next AMX system tick  */

   if (cjevcreate(&motorgroup, "EVMT", motorstatus) == CJ_EROK) {
            if (cjtmcreate(&timerid, "TMMT",
                                 (CJ_TMRPROC)tpmotor,
                                 cjtmconvert(100), NULL
                                 ) == CJ_EROK)

                      cjtmwrite(timerid, 1);   /* Start timer         */
            }
   }
```

```
void CJ_CCPP sttaskA(void)               /* Task A                  */
{
                                         /* Wait forever for motor on*/
   if (cjevwait(motorgroup, MOTORON, MOTORON, CJ_EVOR, 0) == CJ_EROK) {
            :
            Motor is on.
            Process accordingly.
            :
            }
   }



void CJ_CCPP sttaskB(void)               /* Task B                  */
{
                                         /* Wait 5 seconds for motor */
                                         /* on AND at maximum speed  */
   if (cjevwait(motorgroup, MOTOREVT,
                                MOTORON + MOTORMAX, CJ_EVAND,
                                cjtmconvert(5000)
                                ) == CJ_WRTMOUT) {
            :
            Motor not on and up to speed.
            Take recovery action.
            :
            }

   else {
            :
            Motor is on and up to speed.
            Process accordingly.
            :
            }
   }
```

This page left blank intentionally.

8. AMX Mailbox Manager

8.1 Introduction

The AMX Mailbox Manager provides a very flexible, general purpose mechanism for inter process communication and synchronization using messages. In particular, it offers an instant solution to a common problem frequently encountered in real-time applications: one or more processes (producers) having to asynchronously deliver requests for service to one or more servers (consumers) whose identity is unknown to the producer.

For example, assume that two printers are available to print reports and that requests for specific reports originate from an ISP in response to an external action, from a periodic Timer Procedure and from a task which updates a data base. The producers do not know which printer, if any, is free for use at the instant they must initiate their requests. Furthermore, if no printer is free, the ISP and Timer Procedure are unable to wait for a printer to become available. How then to solve the problem?

The Mailbox Manager readily provides the solution. A mailbox is created to act as print request queue. The ISP, Timer Procedure and task send their print requests to the mailbox. Two print server tasks, one for each printer, wait on the mailbox for the next print request. Each print server completes the generation of one report on its printer and then goes back to the mailbox for the next request.

As just illustrated, the AMX Mailbox Manager provides message passing services. The Mailbox Manager permits any task, ISP or Timer Procedure to receive a message. In so doing, it removes the need for the message sender to identify the message receiver.

Messages are delivered to a mailbox message queue in message envelopes. These are the same envelopes that are used by AMX to deliver messages to message exchanges (see Chapters 3.9 and 9).

Any task, ISP, Timer Procedure or Restart Procedure can send a message to a mailbox. The messages are delivered in FIFO order according to their chronological order of transmission. A mailbox differs from a message exchange in that a mailbox has no concept of message priority.

Any task, ISP or Timer Procedure can request a message from a mailbox. Only tasks are allowed to wait for the arrival of a message if none is present in the mailbox when the task makes its request. A task can specify the priority at which it is willing to wait and the maximum time interval which it will wait for a message to arrive.

The task's wait priority determines the order of tasks in the wait queue when more than one task is waiting for a message to arrive at an empty mailbox.

The flexibility of a mailbox comes from the fact that any number of consumers and producers can rendezvous at a mailbox without explicit knowledge of each other. Each consumer and producer only needs to know the id of the mailbox. No consumer or producer owns the mailbox.

Constraints on the use of mailboxes are self-imposed by the system designer. You determine which producers and consumers will use each of your mailboxes. The only restriction imposed by the Mailbox Manager is that only tasks are allowed to wait for a message to arrive at an empty mailbox.

The AMX Mailbox Manager provides the following mailbox services:

cjmbbuild	Create a mailbox (using a definition structure)
cjmbcreate	Create a mailbox (using inline parameters)
cjmbdelete	Delete a mailbox
cjmbflush	Flush all messages or waiting tasks from a mailbox
cjmbsend	Send a message to a mailbox (optional wait for acknowledgement)
cjmbstatus	Get the status of a mailbox
cjmbwait	Get a message from a mailbox (optional timeout)

Your use of the Mailbox Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must provide a hardware clock and include the AMX timing facilities.

Mailboxes can be predefined in your System Configuration Module which is processed by the Mailbox Manager at startup. Mailboxes which are predefined are automatically created by the Mailbox Manager. The mailbox id assigned to each predefined mailbox is stored in a variable which you must provide for that purpose.

Note

If your messages must be ordered by priority, you must use a message exchange.

Using one or the other, but not both, will minimize the code size of your AMX system.

8.2 Mailbox Use

The Mailbox Manager supports any number of mailboxes. The maximum number of mailboxes in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual mailboxes that can be created in your application.

A mailbox must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create mailboxes. It is recommended that only Restart Procedures and tasks be used to create mailboxes.

Create

Mailbox Manager procedure `cjmbbuild` or `cjmbcreate` is used to create a mailbox. The Mailbox Manager allocates a mailbox and returns a mailbox id to the caller. The mailbox id is a handle which uniquely identifies the mailbox. It is the responsibility of the application to keep track of the mailbox id for future reference to the mailbox.

When a mailbox is created, you must specify the maximum number of message envelopes which are allowed to reside in the mailbox. Mailbox depth may range from 1 to 32767.

Mailbox depth has no effect on AMX memory requirements. That is, increasing a mailbox depth does not increase memory needs. The mailbox depth can be used to limit the number of messages allowed to be pending in the mailbox message queue at any instant. All the messages in a mailbox can be flushed with a call to `cjmbflush`.

When a mailbox is created, you can provide a unique 4-character tag to identify the mailbox. The tag can be used subsequently in a call to `cjksfind` to find the mailbox id allocated by the Mailbox Manager to the particular mailbox.

Send

A message is sent to a mailbox with a call to procedure `cjmbsend`. If the specified mailbox is full, the sender will be given an error indication.

A message is a set of parameter bytes located contiguously in memory. The maximum number of parameter bytes in a message is configured by you in your System Configuration Module (see Chapter 15.4). These parameter bytes are completely application dependent. The message size exactly matches that used for message exchange messages.

Messages are sent to mailboxes in AMX message envelopes. The Mailbox Manager gets a free message envelope from the common pool of envelopes maintained by AMX. You must therefore be sure to allocate enough message envelopes to meet the needs of all of your mailbox messages as well as your message exchange messages.

Receive

Any task, ISP or Timer Procedure can get a message from a mailbox by calling procedure `cjmbwait`. If the mailbox has any messages, the message which arrived first will be given to the caller. If the mailbox is empty, the caller has the option of waiting for a message to arrive. ISPs and Timer Procedures must not wait. The caller chooses not to wait by specifying a negative timeout interval in its call to `cjmbwait`. If the caller chooses not to wait, it receives a warning indicating that the mailbox was empty.

Tasks can wait for a message if none is available. The task specifies the interval, measured in system ticks, which it is willing to wait for a message to arrive. An interval of 0 indicates a willingness to wait forever. The task must also specify the priority at which it is prepared to wait, zero being the highest priority.

The Mailbox Manager will add the task to the mailbox's wait queue at the priority the task said it was willing to wait. Thus a task can preempt other tasks already waiting in the queue at lower priority.

When a message finally arrives at the mailbox, the Mailbox Manager will give it to the task, if any, at the head of the mailbox's wait queue. A task switch may occur immediately if the task being given the message is of higher task priority than the currently running task.

If no message arrives at the mailbox before the timeout interval specified by a task expires, the task will be removed from the mailbox wait queue and will resume execution with a timeout indication.

Acknowledge

The Mailbox Manager services include a send and wait feature. A task (and only a task) can send a message to a mailbox and wait for acknowledgement of receipt of the message from the mailbox by some other task. Such messages are called **ack-back** messages. Message acknowledgement is reserved for task to task synchronization.

A task calls `cjmbsend` to send an ack-back message to a mailbox. A parameter in the call indicates that the sender wishes to wait for message acknowledgement. There is no timeout on the wait; the task will wait forever for its acknowledgement.

The ack-back message is received by another task with a call to `cjmbwait`. Once the task has decoded the message and acted upon the sender's request, it can call AMX procedure `cjtkmsgack` with a positive integer answer-back status acknowledging its receipt of the message. If the waiting message sender task is of higher priority than the task which received and acknowledged the message, there will be an immediate task switch. The sending task resumes execution with the receiving task's answer-back status available for interpretation.

The application tasks must know which messages require acknowledgement. This can be accomplished either by task design or by message content. AMX helps a little. If a task ends without acknowledging an ack-back message which it has retrieved from a mailbox, AMX automatically acknowledges the message with an answer-back status of `CJ_EROK`.

The Mailbox Manager prohibits a task which has retrieved an ack-back message from fetching another message from any mailbox or message exchange until the task acknowledges the ack-back message in its possession.

Flush

At any time, a task (and only a task) can call `cjmbflush` to flush a mailbox. The Mailbox Manager will flush all messages, if any, from the mailbox freeing the message envelopes for reuse. If an ack-back message is flushed, the task waiting for the acknowledgement is immediately allowed to resume execution with a warning that its message was flushed.

If the mailbox is empty, tasks, if any, waiting at the mailbox for a message are immediately allowed to resume with a warning that they did not get a message because the mailbox was flushed. An immediate task switch will occur if any of the flushed tasks is of higher priority than the task requesting the flush.

Status

At any time, a task can call `cjmbstatus` to get the status of a mailbox. The status provides the mailbox message depth and a count of the number of messages in the mailbox or the number of tasks waiting for a message from the mailbox.

Delete

When a mailbox is no longer needed, it can be deleted with a call to `cjmbdelete`. The Mailbox Manager will reject the attempt to delete the mailbox if any task is waiting on the mailbox wait queue or if any messages remain in the mailbox. When the Mailbox Manager deletes the mailbox, it marks the mailbox as invalid such that any subsequent reference to the mailbox will be rejected.

You must be absolutely sure that no producer or consumer is referencing the mailbox just as you go to delete it. Be aware that the deleted mailbox id may be immediately reused by AMX for some other purpose.

8.3 Mailbox Application

The following example, coded in C, is provided to illustrate the use of the AMX Mailbox Manager.

The example shows an ISP sending messages to a mailbox. Two tasks, A and B, accept and service the messages from the ISP on a first-come first-served basis. The tasks expect messages to arrive within 5 seconds.

A Restart Procedure creates the mailbox with room for ten messages. In our example, the tasks and ISP assume a valid mailbox id has been provided in variable `mailbox`.

The creation and starting of tasks A and B is outside the scope of this example. It is assumed that if the Restart Procedure is unable to create the mailbox, the tasks will not be started and the ISP interrupt source will be inhibited.

Note that the two tasks, `sttaskA` and `sttaskB`, share a common re-entrant task body, `sttask`. In practice, procedure `sttaskB` could be eliminated and Task B could be created with `sttaskA` as the task start address. The example is coded to more clearly illustrate that two tasks actually exist.

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

static CJ_ID mailbox;                      /* Mailbox id               */

struct usermsg {                           /* Message structure        */
   int      msgtype;                       /* Application message type */
   void     *datapntr;                     /* Application data pointer */
   };

void CJ_CCPP rrmsg(void)                   /* Restart Procedure        */
{
   cjmbcreate(&mailbox, "MBOX", 10);       /* Create mailbox           */
   }
```

```
void CJ_CCPP sttask(void)                /* Common task body          */
{
   union {
            struct cjxmsg maxmsg;        /* Maximum message size      */
            struct usermsg umsg;         /* User message              */
            } msg;

   int      status;

                                         /* Wait 5 sec for message    */
                                         /* Wait at priority 0        */
   status = cjmbwait(mailbox, &msg, 0, cjtmconvert(5000));

   if (status == CJ_EROK) {
            :
            Process the message in union msg.umsg
            :
            }
   else if (status == CJ_WRTMOUT) {
            :
            Process timeout - no message in 5 seconds
            :
            }
   else {
            :
            Process some other error condition
            :
            }
   }


void CJ_CCPP sttaskA(void)               /* Task A                    */
{
   for (;;) sttask();                    /* Do sttask forever         */
   }


void CJ_CCPP sttaskB(void)               /* Task B                    */
{
   for (;;) sttask();                    /* Do sttask forever         */
   }


void CJ_CCPP inthand(void)               /* Interrupt Handler         */
{
   struct usermsg ispmsg;                /* ISP message               */
   :
   Clear interrupt source
   Construct message in structure ispmsg
   :
                                         /* Send message              */
   cjmbsend(mailbox, &ispmsg, CJ_NO);    /* Do not wait for ack       */
   }
```

This page left blank intentionally.

9. AMX Message Exchange Manager

9.1 Introduction

The AMX Message Exchange Manager provides a very flexible, general purpose mechanism for inter process communication and synchronization using prioritized messages. In particular, it offers an instant solution to a common problem frequently encountered in real-time applications: one or more processes (producers) having to asynchronously deliver requests of varying priorities for service to one or more servers (consumers) whose identity is unknown to the producer.

For example, assume that two printers are available to print reports and that requests for specific reports originate from a periodic Timer Procedure, from a data acquisition task and from a data base update task. The producers do not know which printer, if any, is free for use at the instant they must initiate their requests. Furthermore, if no printer is free, the data acquisition task and Timer Procedure are unable to wait for a printer to become available. The data acquisition task must be able to inject high priority print requests when errors are detected. How then to solve the problem?

The Message Exchange Manager readily provides the solution. A message exchange is created to act as a prioritized print request queue. The data acquisition task, Timer Procedure and data base task send their print requests to the message exchange. Two print server tasks, one for each printer, wait on the message exchange for the next print request. Each print server completes the generation of one report on its printer and then goes back to the exchange for the next request.

As just illustrated, the AMX Message Exchange Manager provides message passing services. The Message Exchange Manager permits any task, ISP or Timer Procedure to receive a message. In so doing, it eliminates any need for the message sender to identify the message receiver.

The Message Exchange Manager uses a message exchange to deliver messages. A message exchange consists of four message queues into which messages can be deposited. The message queues are ordered according to priority (0, 1, 2 or 3), message queue 0 being of highest priority.

Messages are delivered to the message queues in a message exchange in message envelopes. These are the same envelopes that are used by AMX to deliver messages to mailboxes (see Chapters 3.9 and 8).

Any task, ISP, Timer Procedure or Restart Procedure can send a message to a message exchange. The sender indicates the priority of its message (0 to 3) thereby identifying the message queue into which it will be delivered.

Any task, ISP or Timer Procedure can request a message from a message exchange. Only tasks are allowed to wait for the arrival of a message if none is present in the message exchange when the task makes its request. A task can specify the priority at which it is willing to wait and the maximum time interval which it will wait for a message to arrive.

The task's wait priority is not to be confused with the message queue priority. The message queue priority determines the priority ordering of messages in the message exchange when no task is waiting for a message. The task's wait priority determines the order of tasks in the wait queue when more than one task is waiting for a message to arrive at an empty message exchange.

The flexibility of a message exchange comes from the fact that any number of consumers and producers can rendezvous at a message exchange without explicit knowledge of each other. Each consumer and producer only needs to know which message exchange to use. No consumer or producer owns the message exchange.

Constraints on the use of message exchanges are self-imposed by the system designer. You determine which producers and consumers will use each of your message exchanges. The only restriction imposed by the Message Exchange Manager is that only tasks are allowed to wait for a message to arrive at an empty message exchange.

The AMX Message Exchange Manager provides the following message exchange services:

cjmxbuild	Create a message exchange (using a definition structure)
cjmxcreate	Create a message exchange (using inline parameters)
cjmxdelete	Delete a message exchange
cjmxflush	Flush all messages or waiting tasks from a message exchange
cjmxsend	Send a message to a message exchange (optional wait for acknowledgement)
cjmxstatus	Get the status of a message exchange
cjmxwait	Get a message from a message exchange (optional timout)

Your use of the Message Exchange Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must provide a hardware clock and include the AMX timing facilities.

Message exchanges can be predefined in your System Configuration Module which is processed by the Message Exchange Manager at startup. Message exchanges which are predefined are automatically created by the Message Exchange Manager. The message exchange id assigned to each predefined message exchange is stored in a variable which you must provide for that purpose.

Note

If your messages do not have to be ordered by priority, use a mailbox instead of a message exchange.

Using one or the other, but not both, will minimize the code size of your AMX system.

9.2 Message Exchange Use

The Message Exchange Manager supports any number of message exchanges. The maximum number of message exchanges in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual message exchanges that can be created in your application.

A message exchange must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create exchanges. It is recommended that only Restart Procedures and tasks be used to create message exchanges.

Create

Message Exchange Manager procedure *`cjmxcreate`* is used to create a message exchange. The Message Exchange Manager allocates a message exchange and returns a message exchange id to the caller. The exchange id is a handle which uniquely identifies the message exchange. It is the responsibility of the application to keep track of the message exchange id for future reference to the exchange.

When a message exchange is created, you must specify the maximum number of message envelopes which are allowed to reside in each of the message exchange's four message queues. Message queue depths may range from 0 to 32767. If a particular queue is not used, set that queue's depth to zero. At least one message queue must have a non-zero depth.

Message queue depth has no effect on AMX memory requirements. That is, increasing a message queue depth does not increase memory needs. The message queue depths can be used to limit the number of messages allowed to be pending at the message exchange at any instant. All the messages in a message exchange can be flushed with a call to *`cjmxflush`*. Messages of all priorities will be flushed.

When a message exchange is created, you can provide a unique 4-character tag to identify the message exchange. The tag can be used subsequently in a call to *`cjksfind`* to find the message exchange id allocated by the Message Exchange Manager to the particular message exchange.

When the Message Exchange Manager creates a message exchange, it sets all of the message exchange's message queues empty.

Send

A message is sent to a message exchange with a call to procedure *`cjmxsend`*. The sender must specify the message priority (0 highest; 3 lowest) thereby indicating the message queue into which the message will be delivered. If the specified message queue is full or has been defined to have a depth of zero, the sender will be given an error indication.

A message is a set of parameter bytes located contiguously in memory. The maximum number of parameter bytes in a message is configured by you in your System Configuration Module (see Chapter 15.4). These parameter bytes are completely application dependent. The message size exactly matches that used for mailbox messages.

Messages are sent to message exchanges in AMX message envelopes. The Message Exchange Manager gets a free message envelope from the common pool of envelopes maintained by AMX. You must therefore be sure to allocate enough message envelopes to meet the needs of all of your message exchange messages as well as your task messages.

Receive

Any task, ISP or Timer Procedure can get a message from a message exchange by calling procedure *cjmxwait*. If the message exchange has any messages, the highest priority message (0 highest; 3 lowest) which arrived first will be given to the caller. If the message exchange is empty, the caller has the option of waiting for a message to arrive. ISPs and Timer Procedures must not wait. The caller chooses not to wait by specifying a negative timeout interval in its call to *cjmxwait*. If the caller chooses not to wait, it receives a warning indicating that the message exchange was empty.

Tasks can wait for a message if none is available. The task specifies the interval, measured in system ticks, which it is willing to wait for a message to arrive. An interval of 0 indicates a willingness to wait forever. The task must also specify the priority at which it is prepared to wait, zero being the highest priority.

The Message Exchange Manager will add the task to the message exchange's wait queue at the priority the task said it was willing to wait. Thus a task can preempt other tasks already waiting in the queue at lower priority.

When a message finally arrives at the message exchange, the Message Exchange Manager will give it to the task, if any, at the head of the message exchange's wait queue. A task switch may occur immediately if the task being given the message is of higher task priority than the currently running task.

If no message arrives at the message exchange before the timeout interval specified by a task expires, the task will be removed from the message exchange wait queue and will resume execution with a timeout indication.

Acknowledge

The Message Exchange Manager services include a send and wait feature. A task (and only a task) can send a message to a message exchange and wait for acknowledgement of receipt of the message from the message exchange by some other task. Such messages are called **ack-back** messages. Message acknowledgement is reserved for task to task synchronization.

A task calls *cjmxsend* to send an ack-back message to a message exchange. A parameter in the call indicates that the sender wishes to wait for message acknowledgement. There is no timeout on the wait; the task will wait forever for its acknowledgement.

The ack-back message is received by another task with a call to *cjmxwait*. Once the task has decoded the message and acted upon the sender's request, it can call AMX procedure *cjtkmsgack* with a positive integer answer-back status acknowledging its receipt of the message. If the waiting message sender task is of higher priority than the task which received and acknowledged the message, there will be an immediate task switch. The sending task resumes execution with the receiving task's answer-back status available for interpretation.

The application tasks must know which messages require acknowledgement. This can be accomplished either by task design or by message content. AMX helps a little. If a task ends without acknowledging an ack-back message which it has retrieved from a message exchange, AMX automatically acknowledges the message with an answer-back status of *CJ_EROK*.

The Message Exchange Manager prohibits a task which has retrieved an ack-back message from fetching another message from any mailbox or message exchange until the task acknowledges the ack-back message in its possession.

Flush

At any time, a task (and only a task) can call *cjmxflush* to flush a message exchange. The Message Exchange Manager will flush all messages, if any, from all message queues freeing the message envelopes for reuse. If an ack-back message is flushed, the task waiting for the acknowledgement is immediately allowed to resume execution with a warning that its message was flushed.

If the message exchange is empty, tasks, if any, waiting at the message exchange for a message are immediately allowed to resume with a warning that they did not get a message because the message exchange was flushed. An immediate task switch will occur if any of the flushed tasks is of higher priority than the task requesting the flush.

Status

At any time, a task can call *cjmxstatus* to get the status of a message exchange. The status provides the depth of each of the exchange's message queues and a count of the number of messages in each of the message queues or the number of tasks waiting for a message from the message exchange.

Delete

When a message exchange is no longer needed, it can be deleted with a call to *cjmxdelete*. The Message Exchange Manager will reject the attempt to delete the message exchange if any task is waiting on the message exchange wait queue or if any messages remain in any of the exchange's message queues. When the Message Exchange Manager deletes the message exchange, it marks the message exchange as invalid such that any subsequent reference to the message exchange will be rejected.

You must be absolutely sure that no producer or consumer is referencing the message exchange just as you go to delete it. Be aware that the deleted message exchange id may be immediately reused by AMX for some other purpose.

9.3 Message Exchange Application

The following example, coded in C, is provided to illustrate the use of the AMX Message Exchange Manager.

This example illustrates a solution to the problem posed in the introduction (Chapter 9.1). Two message processing tasks, A and B, accept and service the messages from the data acquisition and data base tasks and the Timer Procedure. The processing tasks expect messages to arrive within 5 seconds.

A Restart Procedure creates the message exchange with message queues 0, 2 and 3, each of depth 10. In our example, the tasks and Timer Procedure assume a valid message exchange id has been provided in variable *msgexch*. The Restart Procedure also creates and starts the Timer Procedure.

The creation and starting of tasks A and B and the creation of the data acquisition and data base tasks is outside the scope of this example. It is assumed that if the Restart Procedure is unable to create the message exchange, the tasks will not be started.

Since we do not know how the data acquisition task and the data base update task were created, we will assume that their task ids are available in global variables as would be the case if the AMX Configuration Builder had been used to create the tasks.

The Timer Procedure triggers the data acquisition task once a second and sends a time-stamp message to the message exchange at the highest priority, 0.

The data acquisition task samples the data, triggers the data base update task and sends a data sample message to the message exchange at medium priority 2.

The data base update task processes the sampled data, updates the data base and sends a data update message to the message exchange at low priority 3.

Note that the two tasks, `sttaskA` and `sttaskB`, share a common re-entrant task body, `sttask`. In practice, procedure `sttaskB` could be eliminated and Task B could be created with `sttaskA` as the task start address. The example is coded to more clearly illustrate that two tasks actually exist.

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

extern CJ_ID dactid;                    /* Data acquisition task id     */
extern CJ_ID dbtid;                     /* Data base update task id     */

static CJ_ID msgexch;                   /* Message exchange id          */

struct appmsg {                         /* Message structure            */
   int      msgtype;                    /* Application message type     */
   long     data;                       /* Data sample                  */
   void     *dbp;                       /* Pointer to data base         */
   };

union msgu {
   struct cjxmsg dummy;                 /* Maximum message size         */
   struct appmsg umsg;                  /* User message                 */
   };


                                        /* Timer Procedure              */
void CJ_CCPP tmrproc(CJ_ID tmrid, void *unused)
{
   union msgu msgbuf;                   /* Message buffer               */

   :
   Construct message in msgbuf.umsg
   :
                                        /* Send message at priority 0  */
                                        /* No ack-back                  */
   cjmxsend(msgexch, &msgbuf, CJ_NO, 0);

   cjtktrigger(dactid);                 /* Trigger data acquisition task*/
   }


void CJ_CCPP rrmsg(void)                /* Restart Procedure            */
{
   CJ_ID    tmrid;                      /* Timer id                     */

   if (cjtmcreate(&tmrid, "TIMR",       /* Create periodic 1 sec timer */
                  (CJ_TMRPROC)tmrproc,
                  cjtmconvert(1000), NULL
                  ) == CJ_EROK)


            cjtmwrite(tmrid, 1);        /* Start timer immediately      */

                                        /* Create message exchange      */
   cjmxcreate(&msgexch, "MSGX", 10, 0, 10, 10);
   }
```

```
void CJ_CCPP sttask(void)                /* Common task body         */
{
   union msgu msgbuf;                    /* Message buffer           */
   int      status;

                                         /* Wait at priority 0 for   */
                                         /* up to 5 sec for message  */
   status = cjmxwait(msgexch, &msgbuf, 0, cjtmconvert(5000));

   if (status == CJ_EROK) {
            :
            Process the message in msgbuf.umsg
            :
            }

   else if (status == CJ_WRTMOUT) {
            :
            Process timeout - no message in 5 seconds
            :
            }

   else {
            Message retrieval failed
            Process some other error condition
            }
   }


void CJ_CCPP sttaskA(void)               /* Task A                   */
{
   for (;;) sttask();                    /* Do sttask forever        */
   }


void CJ_CCPP sttaskB(void)               /* Task B                   */
{
   for (;;) sttask();                    /* Do sttask forever        */
   }
```

```
void CJ_CCPP dactask(void)              /* Data acquisition task    */
{
   union msgu msgbuf;                   /* Message buffer           */

   :
   Perform data acquisition functions
   :
   if (no_error) {
            :
            Construct message in msgbuf.umsg
            :
                                        /* Send message at priority 3*/
                                        /* No ack-back              */
            cjmxsend(msgexch, &msgbuf, CJ_NO, 3);
            :
            }

   else {                               /* Error has occurred!      */
            :
            Construct message in msgbuf.umsg
            :
                                        /* Send error message at    */
                                        /* priority 0; no ack-back  */
            cjmxsend(msgexch, &msgbuf, CJ_NO, 0);
            }

   cjtktrigger(dbtid);                  /* Trigger data base task   */
   }


void CJ_CCPP dbasetask(void)            /* Data base update task    */
{
   union msgu msgbuf;                   /* Completion message       */

   :
   Perform data base update functions
   :
   :
   Construct message in msgbuf.umsg
   :
                                        /* Send completion message at*/
                                        /* priority 2; no ack-back  */
   cjmxsend(msgexch, &msgbuf, CJ_NO, 2);
   }
```

This page left blank intentionally.

10. AMX Buffer Manager

10.1 Introduction

The AMX Buffer Manager simplifies the management of memory buffers in a real-time system. It provides a general mechanism for the allocation and control of fixed size buffers.

The AMX Buffer Manager provides fast, efficient access to multiple pools of buffers, each buffer representing a fixed size block of memory. This form of memory management meets the requirements of most typical applications and is best suited for real-time use in which memory block availability must be predictable and in which the penalties for memory fragmentation cannot be tolerated.

You define a set of buffer pools, each pool containing a set of buffers of uniform size. Buffer Manager procedures are called to obtain a buffer from a particular pool and to release it back to the pool when it is no longer required.

When released, the buffer is automatically returned by the Buffer Manager to the pool to which the buffer belongs. Buffer ownership can be increased so that more than one task can simultaneously own a shared buffer. Special facilities are provided to assure that if a buffer is owned by more than one task, it is only returned to its pool when the slowest owner finally releases it.

Speed of execution is not dependent on the number of pools or buffers. The number of pools and number of buffers in a pool which can be supported is limited only by memory availability.

The AMX Buffer Manager provides the following buffer management services:

cjbmbuild	Create a buffer pool (using a definition structure)
cjbmcreate	Create a buffer pool (using inline parameters)
cjbmdelete	Delete a buffer pool
cjbmfree	Release a buffer
cjbmget	Get a buffer from a specific buffer pool (optional timeout)
cjbmid	Get buffer pool id for a specific buffer
cjbmsize	Get size of a buffer
cjbmstatus	Get status of a buffer pool
cjbmuse	Add to buffer use count

Your use of the Buffer Manager is optional. If you intend to use it you must indicate so in your System Configuration Module. You must provide a hardware clock and include the AMX timing facilities.

Buffer pools can be predefined in your System Configuration Module which is processed by the Buffer Manager at startup. Buffer pools which are predefined are automatically created by the Buffer Manager. The buffer pool id assigned to each predefined buffer pool is stored in a variable which you must provide for that purpose.

10.2 Buffer Pool Use

The Buffer Manager supports any number of pools of buffers. The maximum number of buffer pools in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual buffer pools that can be created in your application.

A pool of buffers consists of any number of buffers of a uniform size measured in bytes. Any buffer size which is a multiple of 4 and greater than or equal to the target dependent minimum is allowed. The minimum buffer size, *CJ_MINBFS* bytes, is defined in AMX header file *CJZZZKC.H*.

Create Buffer Pool

A buffer pool must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create buffer pools. It is recommended that only Restart Procedures and tasks be used to create buffer pools.

Buffer Manager procedure *cjbmbuild* or *cjbmcreate* is used to create a buffer pool. The Buffer Manager allocates a buffer pool and returns a buffer pool id to the caller. The pool id is a handle which uniquely identifies the buffer pool. It is the responsibility of the application to keep track of the pool id for future reference to the buffer pool. Given the pointer to a buffer within the pool, the buffer pool id can be obtained by a call to *cjbmid*.

When a buffer pool is created, you must specify the following parameters: the number of buffers in the pool, the size of each buffer and a pointer to RAM storage for all of the buffers in the pool.

When a buffer pool is created, you can provide a unique 4-character tag to identify the buffer pool. The tag can be used subsequently in a call to *cjksfind* to find the buffer pool id allocated by the Buffer Manager to the particular buffer pool.

When the Buffer Manager creates a buffer pool, it subdivides the allocated RAM storage into the required number of buffers.

All buffers in the pool are linked together on a free list. This free list is internal to the Buffer Manager, hidden from view. Associated with (but not part of) each buffer is a use count, also internal to the Buffer Manager. The use count is set to zero for each buffer to show that it is not in use.

Get Buffer

Once a buffer pool has been created, you may call procedure *`cjbmget`* to get a buffer from the pool. The Buffer Manager unlinks a buffer from the pool's free list, sets the associated buffer use count to one and returns a pointer to the first byte of the buffer. You may then store and retrieve any data in the buffer as desired.

If there are no free buffers available in the pool when a request to get a buffer from the pool is made, the caller has the option of waiting for a buffer to become available. Restart Procedures, ISPs and Timer Procedures must not wait. The caller chooses not to wait by specifying a negative timeout interval in its call to *`cjbmget`*. If the caller chooses not to wait, it receives a warning indicating that no buffers are available.

Tasks can wait for a buffer if none is available. The task specifies the interval, measured in system ticks, which it is willing to wait for a buffer to become available. An interval of 0 indicates a willingness to wait forever. The task must also specify the priority at which it is prepared to wait, zero being the highest priority.

The Buffer Manager will add the task to the buffer pool wait queue at the priority the task said it is willing to wait. Thus a task can preempt other tasks already waiting in the queue at lower priority.

Free Buffer

When the buffer is no longer required, you can call procedure *`cjbmfree`* to release the buffer. You indicate the buffer to be released by specifying the pointer to the first byte of the buffer, the same pointer that was received when you originally acquired the buffer. The Buffer Manager decrements the use count associated with the buffer by one. If the use count becomes zero, the buffer is linked to the free list of the pool to which it belongs.

Use Count

Once you own a buffer, you may call procedure *`cjbmuse`* to increase the use count. Recall that when a buffer is first obtained, the use count is set to one. If the use count is then increased by one, the buffer will have to be released twice before it becomes free. In certain applications such as described in Chapter 10.3, this feature is essential.

Status

You may also call procedure *`cjbmsize`* to obtain the size of the buffer. Because the buffer was obtained from a pool with buffers of known size, this call is usually unnecessary. However, in applications in which the current owner of a buffer did not actually acquire the buffer from the Buffer Manager, it is convenient to be able to determine the buffer size.

Given a buffer allocated by the Buffer Manager, you can find its buffer pool id with a call to procedure *`cjbmid`*.

The status of a particular buffer pool can be determined with a call to procedure *`cjbmstatus`*. The status provides the buffer size, the total number of buffers and the number of free buffers, if any, or the number of tasks, if any, waiting for buffers.

Delete

If at some point, there is no longer a need for any of the buffers in the buffer pool, the entire buffer pool can be deleted with a call to procedure *`cjbmdelete`*. It is your responsibility to assure that none of the buffers in the pool are still being used at the time you delete the pool. Once a buffer pool has been deleted, all of the RAM storage provided by you when the buffer pool was created is available for reuse by the application.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the buffer pool just as you go to delete it. Be aware that the deleted buffer pool id may immediately be reused by AMX for some other purpose.

10.3 Buffer Applications

Consider the following example. A process control system using AMX has a printer on which errors and status messages are to be logged. These messages are generated by several tasks as they perform their process control functions. These tasks must not wait for the printer because they have more important work to do. Furthermore, each message size may exceed the standard AMX message size.

A printer task receives the messages from the other tasks and outputs them to the printer. The problem is to get these messages from the process control tasks to the printer task.

The Buffer Manager provides an ideal solution. A pool of buffers is defined and dedicated to error logging. As each process control task detects an error, it calls the Buffer Manager to obtain a buffer, fills the buffer with an appropriate error message and makes an AMX *`cjmbsend`* call, sending the buffer pointer in an AMX message envelope to a printer mailbox.

The printer task permanently waits for these AMX messages to arrive at the printer mailbox. The printer task extracts the buffer pointer from the AMX message, outputs the contents of the buffer to the printer, calls the Buffer Manager to release the buffer and ends. Because the process control tasks need never wait for the printer task or printer, they are always available to perform their intended functions.

Now consider the following addition to this example. Suppose it is desired to log messages on both a display and printer. There are several ways this could be done.

1. Two buffers could be obtained from the Buffer Manager. The same message would be copied into both buffers. One would be sent to a printer mailbox, the other to a display mailbox.

2. A single buffer could be obtained from the Buffer Manager. It would be filled with a message and sent to the display mailbox. The display task would output the message to the display and send the buffer to the printer mailbox. The printer task would output the message to the printer and release the buffer.

3. A single buffer could be obtained from the Buffer Manager. The message would be copied into the buffer. The Buffer Manager would be called to increase the buffer use count by one, for a total use count of two. The buffer pointer would then be sent to both the printer mailbox and the display mailbox. Both tasks would output the message to their output device and release the buffer. When the slowest of these two tasks released the buffer, the use count would be zero and the buffer would be free.

These solutions each have their own advantages and disadvantages. Method 1 requires twice as many buffers as the other methods and requires extra processor time to copy the message twice. Method 2 cannot display the message simultaneously on both the CRT and printer. Also, method 2 requires the display task to know about the printer mailbox. Method 3 is the most desirable.

Can method 3 be improved? Having the process control tasks know about the display and printer makes future system modification difficult. Instead, a single reentrant message routing procedure could be added to process all messages and determine if they should be sent to the display mailbox or printer mailbox or both. If the message routing procedure decided to send the buffer to both, it would call the Buffer Manager to increase the buffer use count before passing the buffer on. As in method 3, the display and printer tasks would only have to output the buffer, release it and then end. The process control tasks would only have to get a buffer from the Buffer Manager, fill it with a message and call the message routing procedure to send it to the appropriate mailboxes.

What if the messages have to be sorted by priority? Just use message exchanges instead of mailboxes.

10.4 Buffer Manager Caveats

Although the Buffer Manager attempts to check as many error conditions as possible, it cannot protect against a bad system design. However, if a little care is taken during system design, the Buffer Manager can help make a system more reliable than one that uses an ad hoc buffer management method.

The most important concept to understand about the Buffer Manager is **buffer ownership**. The Buffer Manager owns all free buffers. You must never modify these in any way.

Once you get a buffer (with a call to the Buffer Manager), you own the buffer. You may modify the contents of the buffer while it is owned. Once you release the buffer or pass it to another concurrently executing task or Interrupt Service Procedure, then the buffer is no longer owned by you and may not be modified or released or have its use count altered by you in any way.

If the use count is increased by the owner of a buffer, then several simultaneous owners are allowed. However, when there are several concurrently executing owners, the buffer content must not be modified unless your design is such that each owner can write to its own private portion of the buffer. The only permitted operations are reading the buffer and releasing it. Each owner should either release the buffer or pass the ownership on to another routine that will release it.

These restrictions are characteristic of message passing between concurrently executing routines in general, not just of the AMX Buffer Manager. To illustrate the necessity of these restrictions, a few examples of incorrect usage follow.

1. Task A gets a buffer, fills it, passes it to Task B and releases it. Task B attempts to use the data in the buffer.

   The problem is that Task A released the buffer after it passed ownership to Task B. Thus Task B will receive a buffer which it does not own. If Task B modifies the buffer, then the Buffer Manager may produce unpredictable results the next time it tries to use the buffer.

   The solution is to have Task B release the buffer when it is finished with it. Task B may do this because it owns the buffer after it receives it from Task A.

2. Task A gets a buffer, fills it, passes it to Task B, increments the use count by one and passes the buffer to task C. Tasks B and C use the buffer and release it.

   The problem is that Task A has passed ownership to Task B without first retaining the ownership by increasing the use count. It then increases the use count of a buffer it doesn't own. In the meantime, Task B may have finished using the buffer and released it.

   The solution is to have Task A expand the ownership by increasing the use count before passing the buffer to Task B. Task A then retains one half of the ownership and may pass this ownership on to Task C.

This page left blank intentionally.

11. AMX Memory Manager

11.1 Introduction

The AMX Memory Manager simplifies the management of memory in an AMX system. It provides a general mechanism for the dynamic allocation and control of memory and is specifically designed for use in a multitasking environment.

Multitasking adds particular difficulties to the general problem of memory allocation. For example, high level language memory management procedures, such as C's `malloc`, `calloc` or `free`, cannot be called from concurrently executing tasks since the procedures are usually not reentrant.

The Memory Manager resolves these difficulties. The Memory Manager's general allocation procedures allow the creation of memory pools from which concurrently executing tasks can allocate and free blocks of memory.

You define a set of memory pools, each pool containing a set of one or more sections of contiguous memory. Memory Manager procedures are called to obtain a memory block of any size from a particular memory pool and to release it back to the pool when it is no longer required.

When released, the memory is automatically returned by the Memory Manager to the memory pool to which it belongs. Memory block ownership can be increased so that more than one task can simultaneously own a shared piece of memory. Special facilities are provided to assure that if a memory block is owned by more than one task, it is only returned to its pool when the slowest owner finally releases it.

The number of memory pools and the amount of memory in each pool is limited only by memory availability.

A particular unique feature of the Memory Manager permits any block of memory (including those acquired from a memory pool controlled by the Memory Manager) to be treated as a memory pool from which smaller private blocks can be dynamically allocated. This feature allows a task which has well defined memory requirements to reserve a memory block for its own private use. The task can then call on the Memory Manager to control the allocation and release of smaller blocks from within the larger private block.

The AMX Memory Manager provides the following memory management services:

`cjmmbuild`	Create a memory pool (using a definition structure)
`cjmmcreate`	Create a memory pool (using inline parameters)
`cjmmdelete`	Delete a memory pool
`cjmmfree`	Release a block of memory
`cjmmget`	Get a block of memory
`cjmmid`	Get memory pool id for a specific memory block
`cjmmresize`	Alter the size of a block of memory
`cjmmsection`	Add a section of memory to a memory pool
`cjmmsize`	Get the size of a block of memory
`cjmmuse`	Add to block use count

Your use of the Memory Manager is optional. If you intend to use it you must indicate so in your System Configuration Module.

Memory pools can be predefined in your System Configuration Module which is processed by the Memory Manager at startup. Memory pools which are predefined are automatically created by the Memory Manager. The memory pool id assigned to each predefined memory pool is stored in a variable which you must provide for that purpose.

11.2 Nomenclature

The following nomenclature has been adopted by the Memory Manager.

A **Memory Section** is a contiguous area of Random Access Memory (RAM) which has been assigned by the user to the Memory Manager for use as part of a memory pool.

A **Memory Pool** is a collection of one or more memory sections under the control of the Memory Manager from which tasks can request memory to be allocated for their private use.

A **Memory Block** is a contiguous portion of memory allocated by the Memory Manager from a memory pool for use by a task.

A **Header** (i.e. Memory Block Header) is an area of RAM associated with (but not part of) a memory block. It contains control information which is private to the Memory Manager and not accessible to any task.

A **Block Use Count** is an integer associated with (but not part of) a memory block. It is used by the Memory Manager to keep track of the number of owners of the memory block.

A **Memory Pool Id** is a handle provided by the Memory Manager to identify a memory pool.

11.3 Memory Pool Use

The Memory Manager supports any number of pools of memory. The maximum number of memory pools in a system is defined in your System Configuration Module (see Chapter 15.8). The defined maximum sets an upper limit on the number of actual memory pools that can be created in your application.

A memory pool consists of any number of memory sections of varying sizes measured in bytes. Any memory section size which is a multiple of 4 and greater than or equal to a target dependent minimum is allowed. The minimum memory size, *CJ_MINSMEM* bytes, is defined in AMX header file *CJZZZKC.H*. The minimum is specified in the processor specific AMX Target Guide. Usually a memory pool consists of a single large memory section.

Create

A memory pool must be created by an application before it can be used. Restart Procedures, tasks, ISPs and Timer Procedures can create memory pools. It is recommended that only Restart Procedures and tasks be used to create memory pools.

Memory Manager procedure *cjmmcreate* is used to create a memory pool. The Memory Manager allocates a memory pool and returns a memory pool id to the caller. The pool id is a handle which uniquely identifies the memory pool. It is the responsibility of the application to keep track of the pool id for future reference to the memory pool.

When a memory pool is created, you can assign a memory section to it by indicating the size of the section and providing a pointer to the memory section RAM storage area. Alternatively, a memory pool can be created without any memory and memory sections can be assigned to the memory pool at a later time with calls to procedure *cjmmsection*. There is no restriction on the number of memory sections that can be assigned to a memory pool. Note that the Memory Manager treats two memory sections as disjoint pieces of memory even if the memory for the two sections happens to be physically or logically contiguous.

When a memory pool is created, you can provide a unique 4-character tag to identify the memory pool. The tag can be used subsequently in a call to *cjksfind* to find the memory pool id allocated by the Memory Manager to the particular memory pool.

When the Memory Manager assigns a memory section to a memory pool, it treats the memory section as one large free memory block linked together with other free blocks on a free list. This free list is internal to the Memory Manager, hidden from view.

Get Memory

Once a memory pool has been created, a task can call the Memory Manager procedure *cjmmget* to get a block of memory of any size from the pool. Only tasks are allowed to call the Memory Manager to acquire memory blocks.

If the Memory Manager is able to find a block of memory in the memory pool large enough to meet the caller's requirements, the caller will be given a pointer to the memory block and an indication of the actual size of that block. The block may be marginally larger than the size requested. The memory block use count is set to one.

If the Memory Manager cannot locate a memory block in the memory pool large enough to meet the caller's requirement, it returns an error indication to the caller and identifies the largest block which is available in the pool at that instant. Note, however, that if the calling task immediately requests a block of memory equal to this largest memory block size, the request may still fail because, in a multitasking system, other higher priority tasks may have grabbed some of that memory before the task can make another request to the Memory Manager.

Given a memory block allocated by the Memory Manager, you can find its memory pool id with a call to procedure *cjmmid*.

Free Memory

When use of a memory block is no longer required, the owner can call procedure *cjmmfree* to release the block. The caller specifies the block to be released by passing the Memory Manager the same pointer that it received when the Memory Manager originally allocated the block.

The Memory Manager decrements the memory block's use count and, if the count goes to zero, returns the block to the memory pool to which it belongs. If there are any free memory blocks adjacent to the released block, they are merged with the block being released to form a larger block.

Once a block is released, no task in the system can make reference to it. The block enters the private domain of the Memory Manager.

Use Count

When the Memory Manager allocates a block of memory for the use of a task, it sets the block's use count to one. The block owner may call the Memory Manager procedure *cjmmuse* to increase the use count. If the use count is increased by one, the block will have to be released twice before it becomes free.

The block use count provides the key to memory block ownership. The Memory Manager owns all free blocks in all memory pools. One or more tasks can own an already allocated memory block. This concept of block ownership is the same as that of buffer ownership provided by the AMX Buffer Manager. Refer to Chapters 10.3 and 10.4 for examples of the proper use of this ownership concept.

Size

The Memory Manager procedure *`cjmmsize`* can be used to obtain the size of a particular memory block. This feature can be useful if one task is given ownership of a memory block by another task. The new block owner can check the size of the memory block to be sure that it meets its requirements. If a task corrupts the contents of any memory location outside the bounds of the memory block, the effects are unpredictable and potentially disastrous.

Resize

The Memory Manager procedure *`cjmmresize`* can be used to grow or shrink the size of a memory block. If a task owning a memory block finds that its memory block is too small or too large, the task can call *`cjmmresize`* to try to adjust the size of the block to meet its needs.

If the memory block is to be shrunk, the Memory Manager carves a piece from the high end of the block and adds the new fragment to the memory pool's free list. The fragment is merged with an adjacent free memory block, if one exists, to form a larger free block.

A memory block can only grow upward in memory. If the memory block is to grow in size, the Memory Manager checks to see if the memory block immediately higher in memory than the caller's block is free. If that free block, merged with caller's block, meets the growth requirement, the blocks are merged to form a larger block and then shrunk to meet the caller's specification. If there is no adjacent free block or if one exists but is not large enough to meet the growth requirement, the caller's resize request is rejected with a warning indication.

Note

Because memory blocks are carved from the end (top) of free memory blocks, you will rarely be able to grow a block in size.

Memory Manager procedure *`cjmmsection`* can be used to add more memory sections to a memory pool. The addition of memory sections to a memory pool simply increases the memory available to the Memory Manager for allocation from the pool.

Memory sections cannot be removed from a memory pool.

Delete

If at some point, there is no longer a need for any of the memory in a memory pool, the entire memory pool can be deleted with a call to procedure *`cjmmdelete`*. It is your responsibility to assure that none of the memory in the pool is still being used at the time you delete the pool. Once a memory pool has been deleted, all of the memory section RAM storage assigned by you to the memory pool is available for reuse by the application.

You must be absolutely certain that no task, ISP or Timer Procedure is referencing the memory pool just as you go to delete it. Be aware that the deleted memory pool id may immediately be reused by AMX for some other purpose.

11.4 Private Memory Allocation

A particularly unique feature of the Memory Manager permits any block of memory (including those acquired from the Memory Manager) to be treated as memory from which smaller private blocks can be dynamically allocated.

To use this feature, a task calls procedure *`cjmmcreate`* giving it a pointer to a private area of memory whose access is to be controlled by the Memory Manager. Blocks allocated by procedure *`cjmmget`* are suitable for this purpose. The caller must also specify the size of the memory area provided.

The Memory Manager takes the memory area which it has been given and converts it into a memory pool for the private use of the task. This memory pool is identified by a memory pool id which is returned to the caller. As long as the task does not make the memory pool id public, the pool remains private to the task.

The memory pool id must be used to allocate smaller blocks from the memory pool. Memory Manager procedure *`cjmmget`* is used to acquire a memory block from the private memory pool identified by the private memory pool id. Once a private memory pool has been created, any task having the memory pool id can acquire a memory block from the pool. It is up to the task which created the memory pool to determine which tasks are given the memory pool id.

12. AMX Circular List Manager

12.1 Circular Lists

The AMX Circular List Manager provides a general circular list facility for use by application program modules.

Circular lists must be located in alterable memory (RAM).

A circular list is a data structure used by an application to maintain an ordered list of 8-bit, 16-bit or 32-bit elements. The elements are stored in slots in the list. Each list contains a fixed, user defined number of slots.

The AMX Circular List Manager provides the following list manipulation procedures:

cjclinit	Reset (initialize) a circular list
cjclabl	Add to bottom of circular list
cjclatl	Add to top of circular list
cjclrbl	Remove from bottom of circular list
cjclrtl	Remove from top of circular list

The Circular List Manager procedures are reentrant permitting them to be shared by concurrently executing tasks, ISPs and Timer Procedures.

Examples

Elements can be added to the top and removed from the bottom of a circular list. This makes these lists particularly attractive for character buffering by ISPs. Received characters can be added to the bottom of the circular list and removed from the top. If the process that removes a character decides that it cannot yet handle the character, it can return the character to the top of the circular list.

16-bit lists are useful for handling both a character and its receiver status. When a character is received, the character and its status are added to the bottom of a circular list. The task processing received characters pulls a 16-bit element from the top of the circular list to get each character and its status.

32-bit lists can be used to manage pointers to more complex structures. For instance, a sawmill application might use two lists to keep track of log measurements and resulting lumber counts in a Log Description Block, a private application structure. As logs are measured, their dimensions are inserted into a Log Description Block. A pointer to the block is added to the bottom of a circular list which serves as input to the cutting process. This process removes a pointer to the next available Log Description Block from the top of this list. Once the log is cut, the log's lumber results are added to the block and the block pointer is added to the bottom of another circular list for lumber tally processing.

12.2 Circular List Use

A circular list is created by an application with a call to procedure `cjclinit`. The caller must provide three parameters: the number of slots in the list, the size of each slot (1, 2 or 4 bytes) and a pointer to RAM storage for the circular list. The number of slots in a circular list is limited only by available memory.

The RAM storage area must provide *`(n * s) + sizeof(struct cjxclist)`* bytes where *`n`* is the number of slots and *`s`* is the slot size (1, 2 or 4).

The Circular List Manager creates the list in the storage provided and sets the list empty. The Circular List Manager procedures can then be used to add and remove elements of the defined size at the top and/or bottom of the list.

The Circular List Manager procedures provide the caller with the status of the list following each call. When adding an element to the list, the caller will receive notice if the list is already full. If the element is added to the list, the caller will be notified if the list has just become full.

When removing an element from the list, the caller will receive notice if no element was on the list. If an element is retrieved from the list, the caller will be notified if the list has just gone empty.

12.3 Circular List Structure

Circular lists are application data structures which are only accessible by calls to the AMX Circular List Manager. The internal structure of the list is private to the Circular List Manager.

Lists can be created dynamically or statically by any application program.

The following examples illustrate 8-bit, 16-bit and 32-bit lists created in C. `NSLOT` is defined to be the number of slots in each list.

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

typedef char SLOT1;                       /* 1 byte slot              */
typedef short int SLOT2;                  /* 2 byte slot              */
typedef long SLOT4;                       /* 4 byte slot              */

#define NSLOT 64                          /* 64 slots in the list     */


struct {                                  /* List of 1 byte slots     */
   struct cjxclist header;
   SLOT1    slots[NSLOT];
   } bytelist;

struct {                                  /* List of 2 byte slots     */
   struct cjxclist header;
   SLOT2    slots[NSLOT];
   } shortlist;

struct {                                  /* List of 4 byte slots     */
   struct cjxclist header;
   SLOT4    slots[NSLOT];
   } pntrlist;
```

`Bytelist` is a circular list of `NSLOT` 8-bit elements.
`Shortlist` is a circular list of `NSLOT` 16-bit elements.
`Pntrlist` is a circular list of `NSLOT` 32-bit elements.

This page left blank intentionally.

13. AMX Linked List Manager

13.1 Introduction

The Linked List Manager provides a general set of fast linked list services suitable for use in real-time systems. The Linked List Manager removes the tedium and potential for serious error inherent in many applications in which list maintenance is implemented by each programmer in unique and varying ways.

The Linked List Manager offers a feature not often found in similar utilities. Objects manipulated by the Linked List Manager can concurrently reside on more than one list.

The Linked List Manager also resolves list manipulation races which frequently occur in real-time multitasking applications. The Linked List Manager assures that no list linkages will be corrupted because of races between tasks, ISPs or Timer Procedures attempting to manipulate the same list.

The Linked List Manager supports doubly linked lists in which all objects on a list are linked in forward and backward directions.

The Linked List Manager also supports keyed lists in which the order of objects in the list is determined by an ordering key provided by the application.

The AMX Linked List Manager provides the following list manipulation procedures. The procedures are reentrant permitting them to be shared by concurrently executing tasks, ISPs and Timer Procedures.

`cjlmcreate`	Create an empty list
`cjlminsh`	Insert at the head of a list
`cjlminst`	Insert at the tail of a list
`cjlminsc`	Insert before the specified object on list
`cjlmrmvh`	Remove from the head of a list
`cjlmrmvt`	Remove from the tail of a list
`cjlmrmvc`	Remove specified object from a list
`cjlmhead`	Find current head of a list
`cjlmtail`	Find current tail of a list
`cjlmnext`	Find next object on a list (walk towards tail)
`cjlmprev`	Find previous object on a list (walk towards head)
`cjlminsk`	Insert into a keyed list
`cjlmordk`	Reorder object in a keyed list
`cjlmmerg`	Merge two lists

13.2 Linked Lists

Terminology

A **list header** is a structure provided by the application to be used to anchor a list. The list header is used to identify a list. The content of the list header is private to the Linked List Manager.

An **object** is an application data structure which represents the elements which reside on a list. For example, AMX maintains a list of Task Control Blocks (TCBs). Each TCB is an instance of a data structure representing the state of a task. Hence, the TCB qualifies as an object.

A **list node** is a structure embedded in an object. The list node is used to link the object into a simple doubly linked list. The content of the list node is private to the Linked List Manager.

A **key** is an unsigned integer which is used to determine the order of objects in a keyed list. Each object in a keyed list must have a key. Objects in a keyed list are ordered such that key values are monotonically increasing from the head of the list to the tail of the list.

A **key node** is a structure embedded in an object. The key node is used to link the object into a keyed list. The object's key resides in the key node. The content of the key node is private to the Linked List Manager.

The **head** of a list is the first object on a list. The **tail** of a list is the last object on a list. An empty list has no head or tail. If only one object is on a list, it is both the head and tail of the list.

The **node offset** is the offset (i.e. displacement) into an object at which the list node (or key node) resides. The node offset assigned for a particular list is fixed. Any object which can reside on the list must have a list node (or key node) in the object at the node offset assigned to that list.

Figure 13.2-1 illustrates three doubly linked lists of apples and oranges. All apples and oranges reside on a fruit list. Fresh apples or oranges reside on a fresh list. Rotten apples or oranges reside on a rotten list.

The list objects are assumed to be data structures describing apples and oranges. Each object contains two list nodes: one for use with the fresh list or rotten list, the other for use with the fruit list.

Note that a single list node can be used for the fresh and rotten lists since these lists are mutually exclusive. All fruit list nodes reside at the same node offset in apple or orange objects. All fresh and rotten list nodes reside at the same node offset in apple and orange objects. The Linked List Manager lets you mix objects of different types on the same list as long as each object has a link node in it at the correct node offset for the particular list.

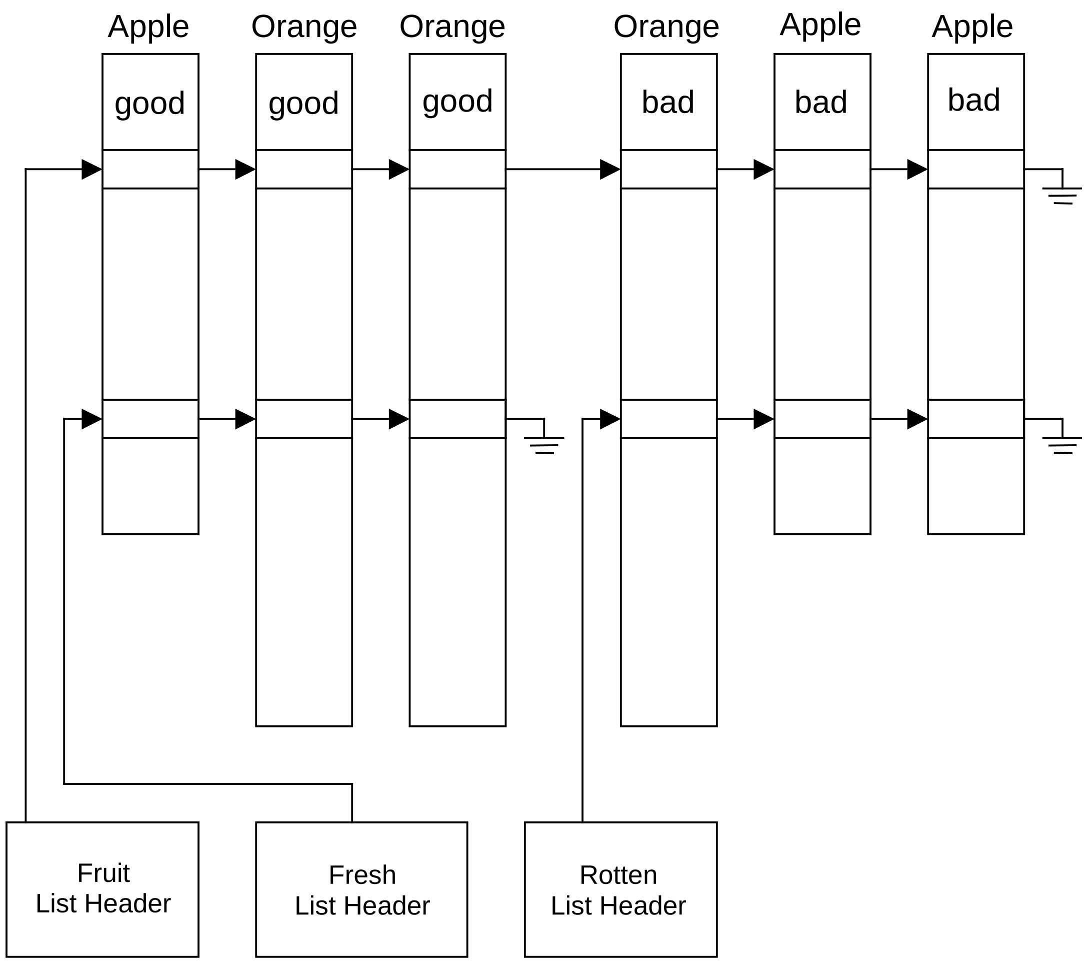


Figure 13.2-1 Doubly Linked Lists

13.3 Linked List Use

A list consists of a list header and objects linked to the list header by list nodes (or key nodes). Storage for the list header must be provided by you. A pointer to the list header acts as the list identifier.

An empty list is created by calling procedure *cjlmcreate* with a pointer to the list header storage. When the list is created, you must specify the node offset (byte displacement) at which link nodes (key nodes) will be found in objects which can reside on the list.

Once a list has been created, any object which has a link node (or key node) at that list's node offset can be added to the list.

Objects can be inserted at or removed from the head or tail of the list. You can always find the head or tail of the list. Given a pointer to any object on the list, you can find the next or previous object, insert another object ahead of it or remove it from the list.

If the objects have a key node at the list's node offset, then the list is a keyed list. All objects on a keyed list must have key nodes at the list's node offset. A new object is added to a keyed list according to the key provided in the call to *cjlminsk*. The object will be inserted into the list after all objects whose key is numerically less than or equal to the specified key. Thus keyed objects are added after all other objects with the same or lesser key.

The position of an object in a keyed list can be altered by calling *cjlmordk* with a new key for the object. The list will be reordered moving the specified object to the correct position in the list according to its new key value. Reordering is usually faster than removing the object with *cjlmrmvc* and reinserting with *cjlminsk*.

The following example coded in C illustrates the use of the Linked List Manager. An object called *uobject* is defined with a key node at offset *keynode* in the object. An array of ten objects is provided. A keyed list *keylist* is created and the ten objects are added to the list in random order. The list is then perused to locate the actual position in the list of the sixth object in the array. The object is removed from the keyed list and added to the tail of a simple doubly linked list called *extlist*.

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

extern unsigned int random();               /* Random number generator  */


struct uobject {
   int      id;                             /* Object identifier        */
   int      data;                           /* Other application data   */
   struct cjxlk keynode;                    /* Key node                 */
   char     moredata[10];                   /* More application data    */
   struct cjxln listnode;                   /* List node                */
   int      lastdata;                       /* Last application data    */
   };


/* Local variables                                                      */

static struct cjxlh keylist;                /* Keyed list header        */
static struct cjxlh extlist;                /* Extraction list          */

#define NUMOBJ 10                           /* Ten objects              */

                                            /* Array of objects         */
static struct uobject objarray[NUMOBJ];
```

```
void CJ_CCPP example(void)
{
   int      i;
   struct uobject *objp;              /* Object pointer             */
   int      nodeofs;                  /* Node offset                */


   nodeofs = (char *)(&objarray[0].keynode) - (char *)&objarray[0];

   cjlmcreate(&keylist, nodeofs);     /* Create empty keyed list    */


   nodeofs = (char *)(&objarray[0].listnode) - (char *)&objarray[0];

   cjlmcreate(&extlist, nodeofs);     /* Create empty extraction list*/


   objp = &objarray[0]; /* First object                             */

   for (i = 1; i <= NUMOBJ; i++) {
            objp->id = i;             /* Insert object id           */
                                      /* Add to list in random order */
            cjlminsk(&keylist, objp++, random());
            }


   objp = cjlmhead(&keylist);         /* Find head of list          */

   while (objp != NULL) {
            if (objp->id == 6)
                        break;        /* Found object of interest   */

            objp = cjlmnext(&keylist, objp);
            }


   if (objp != NULL) {
            cjlmrmvc(&keylist, objp);  /* Remove object from keylist  */
            cjlminst(&extlist, objp);  /* Add to extraction list      */
            }
   }
```

14. Advanced Topics

14.1 Fatal Exit

There are a number of conditions which, if encountered by AMX, are considered to be fatal. Any attempt by AMX to continue execution will lead to unpredictable results at best. All of these conditions cause AMX to force a branch to its fatal exit handler at *cjksfatal* in application module *CJZZZUF.C*.

Insufficient Memory

The most common of these conditions occurs at startup. Many of the AMX managers include a Restart Procedure which is executed by AMX during its startup phase. Some of these managers must initialize a portion of the AMX Data Segment which is private to their needs. If any of these managers find that their needs exceed the size of the data segment provided in your AMX System Configuration Module, AMX takes its fatal exit. To proceed would imply using memory which does not belong to AMX.

Task Error Traps

AMX will take its fatal exit if a processor dependent task trap exception such as division by zero, arithmetic overflow or an array bound error occurs in any Restart Procedure, Interrupt Service Procedure or Timer Procedure.

If any of these errors occur in a task which does not have a task trap handler (see Chapter 4.5) for the corresponding error, AMX will take its fatal exit.

AMX and its managers avoid instructions which produce arithmetic overflow or array bound error exceptions. In the rare occasions when AMX or its managers use division, the division is guaranteed not to produce a divide fault.

Application Faults

If your application encounters conditions which are deemed fatal, you can take the AMX fatal exit by calling procedure *cjksfatal*. You can define your own negative fatal exit codes <= *CJ_FEBASE* to identify your fatal error conditions.

Fatal Exit Procedure

AMX provides a default Fatal Exit Procedure at entry point *cjksfatal* in the AMX application startup module *CJZZZUF.C*.

The default Fatal Exit Procedure disables the external interrupt system and loops forever. Only a hardware reset can be used to recover.

The Fatal Exit Procedure must not make any use of AMX or any of its managers.

In general, there is little that the Fatal Exit Procedure can do. It certainly cannot rectify the situation. Any processing that it does should be done with the interrupts disabled if possible. If not, all interrupt sources should be reset or otherwise inhibited if hardware permits. Once all interrupt sources have been eliminated, the external interrupt system can be enabled. Only then is it acceptable to try restarting your AMX system.

Upon entry to the Fatal Exit Procedure, the following conditions exist:

> Interrupts are immediately disabled.
> All registers are free for use.
> A fatal exit code is provided as a parameter.
> (see *CJ_FExxx* definitions in Appendix B)
> The stack in effect at the time of the fatal exit is in use.

You are free to edit procedure *cjksfatal* in module *CJZZZUF.C* to meet your specific application requirements.

14.2 User Error Procedure

Most AMX procedures return error status to the caller. The error status is a signed integer.

CJ_EROK	= 0	No error
CJ_WRxxx	> 0	Warning: possible fault
CJ_ERxxx	< 0	Error: may be unrecoverable

The defined error codes are summarized in Appendix B.

Before returning an error or warning condition to the caller, AMX calls its User Error Procedure `cjkserror` in AMX application startup module *CJZZZUF.C*. The User Error Procedure receives the AMX error code and the task id of the task executing at the time of the error. The task id will be `CJ_IDNULL` if the error occurred in an ISP call to AMX.

Upon entry to the User Error Procedure, interrupts may be enabled or disabled depending upon conditions when the error was detected.

The User Error Procedure executes in the context of the task, ISP, Timer Procedure, Restart Procedure or Exit Procedure which made the errant AMX call. The User Error Procedure can only make AMX calls which an ISP is allowed to make.

If the User Error Procedure returns to AMX, it must pass the error code back to AMX for return to the caller of the AMX procedure in which the error was detected.

The default User Error Procedure `cjkserror` ignores warnings allowing AMX to return the unaltered warning code back to the AMX procedure caller. When an error is detected, public procedure `cjksbreak` is called allowing error conditions to be isolated for testing purposes. By putting a debug breakpoint on `cjksbreak` you can isolate all occurrences of errors detected by AMX.

In general, there is little that the User Error Procedure can do. It can, however, be extremely useful for locating faults in your application during initial testing. It is also useful for locating infrequently occurring error conditions which are not being checked by your code and hence are going undetected in an otherwise working system.

You are free to edit procedure `cjkserror` in module *CJZZZUF.C* to meet your specific application requirements.

Application Errors

If your application encounters conditions which are deemed serious errors, you can call the AMX error procedure `cjkserror`. You can define your own negative error codes <= *CJ_ERBASE* or positive warning codes >= *CJ_AKBASE* to identify your error conditions. Note that *CJ_AKBASE* is the same positive numbering base reserved for your message acknowledgement codes.

14.3 Task Scheduling Hooks

AMX does not provide direct support for specific hardware extensions such as a math coprocessor or a memory management unit. Instead, AMX allows a set of application procedures to be connected to its Task Scheduler. These procedures can save and restore hardware dependent parameters specific to your application whenever a task switch occurs.

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 assure that they execute as fast as possible.

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

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

Interrupts are disabled and must remain so.
The Task Control Block address is in a register.
The task stack is in use.
A subset of the processor registers is free for use.
All other registers must be preserved.

See the processor specific AMX Target Guide for exact register usage and calling sequence.

Your procedures receive a pointer to the Task Control Block of the task which is being started, ended, suspended or resumed. Your procedures are free to use the task's stack. However, you must not attempt to use this stack to save information. For instance, popping the return address, pushing parameters onto the stack and then returning to AMX is not allowed.

If you have to save information as part of the task's state, you should use the storage in the Task Control Block reserved for the private use of your application (see Chapter 3.12). If necessary, provide an extension to your Task Control Block and install a pointer to the extension in the portion of the Task Control Block reserved for your use.

Once your procedures are installed, you will observe a degradation in the AMX task switching performance. Each call to your procedure will add the setup and calling overhead plus the time it takes your procedure to execute.

14.4 Abnormal Task Termination

A task is a procedure which is called by the AMX Task Scheduler. The task ends execution normally by returning to AMX. AMX provides procedure `cjtkend` which can be used by a task to end execution and return to AMX under circumstances in which its stack is deeply nested.

These two methods by which a task may end execution are considered normal termination. A task has to be running to end in this fashion. No other task can force a task, other than itself, to end.

AMX also provides a set of services which permit one task to force the abnormal termination of another task. These services are not to be treated lightly. Their description has been deferred to this chapter to indicate that they should be rarely invoked.

It is assumed that you have read Chapter 3 of this manual and are familiar with the AMX Task State Diagram presented in Figure 3.2-1.

The ability to arbitrarily terminate a task is one of the most abused privileges afforded by some multitasking systems. The AMX philosophy is to encourage well structured task designs in which there is rarely the need for an uncontrolled task termination. In many cases, a modification to your system design can eliminate the apparent need to arbitrarily terminate a task. However, occasionally system design does demand that an erring task be eliminated for the good of all others. The availability of a controlled task termination can greatly enhance your options in this case.

AMX offers the following task termination services:

`cjtkstop`	Force a task to stop
`cjtkkill`	Kill a task by forcing it to stop
`cjtkdelete`	Delete a task
`cjtkterm`	Enable/disable abnormal task termination

The AMX termination services will not be included in your AMX system if your tasks do not call any of these procedures.

Stop a Task

A task can be stopped. The effect is the same as if the task had just issued a call to `cjtkend` to end its own operation. Only a task which is running, waiting or ready to execute can be stopped. A task can stop itself.

A task that has been stopped does not cease to exist. It simply ends operation. If it has outstanding trigger requests for execution, the task will be allowed to run again.

Kill a Task

A task can be killed. The task is first stopped as just described. All outstanding trigger requests to the task for its execution are purged. The effect is the same as if the task continued to make calls to *cjtkend* to end its operation until finally there were no task execution requests remaining. A task can kill itself.

There are two kill procedures: *cjtkkill* and *cjtkxkill*. Use *cjtkxkill* to kill a message exchange task (see Chapter 14.6). Use *cjtkkill* to kill all other tasks.

A task that has been killed does not cease to exist. Any new requests to trigger the task will be honoured.

Delete a Task

A task can be deleted. A task which is deleted ceases to exist. Its task id becomes invalid and may be reused by AMX for some other purpose. A task can delete itself.

There are two delete procedures: *cjtkdelete* and *cjtkxdelete*. Use *cjtkxdelete* to delete a message exchange task (see Chapter 14.6). Use *cjtkdelete* to delete all other tasks.

Task Termination Procedure

To safeguard against abuses of its task termination services, AMX inhibits any task from being stopped, killed or deleted until the application indicates its willingness to allow the task to be abnormally terminated by calling procedure *cjtkterm*.

When you call *cjtkterm* to allow abnormal termination of a task, you must give AMX a pointer to a Task Termination Procedure to be called by AMX whenever the task is stopped, killed or deleted. You can subsequently inhibit the task's abnormal termination by calling *cjtkterm* with a *CJ_NULLFN* procedure pointer.

Once a task's abnormal termination is enabled, it remains enabled even if the task ends normally. The next time the task executes, it will still be able to be abnormally terminated. The previously defined Task Termination Procedure remains in effect until it is cancelled by calling *cjtkterm* to install a *CJ_NULLFN* pointer.

If a task is stopped or killed, AMX resets its Task Termination Procedure pointer to *CJ_NULLFN* thereby inhibiting any further requests to stop or kill the task until a new Task Termination Procedure has been provided for the task.

If a task which can be abnormally terminated wishes to wait for a buffer, a semaphore, an event group or a message from a mailbox or message exchange, it must temporarily disable task termination, make the wait call and then reinstall its Task Termination Procedure.

A Task Termination Procedure can be coded as a C procedure as illustrated in the following example. The procedure receives an integer reason code indicating whether the task is being stopped, killed or deleted. The mnemonics for these reason codes are provided in the AMX header file *CJZZZSD.H*.

Upon entry to the Task Termination Procedure, the following conditions exist:

> Interrupts are enabled.
> All registers are free for use.
> A reason code is provided as a parameter.
> (see *cjtkterm* description in Chapter 18)
> The task's stack in effect at the time of the termination request is in use.

The Task Termination Procedure must return to AMX. It must not call procedure *cjtkend* to end task execution.

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

void CJ_CCPP termproc(                    /* Task Termination Procedure*/
CJ_ID taskid,                             /* Task id of the task being*/
                                          /* terminated               */
int reason)                               /* Termination reason       */
{
   switch(reason) {

   case     CJ_KCCFSTOP:
            :
            Stop task is occurring
            :
            break;

   case     CJ_KCCFKILL:
            :
            Kill task is occurring
            :
            break;

   case     CJ_KCCFDEL:
            :
            Delete task is occurring
            :
            break;

            } /* end of switch            */
   }
```

Termination Processing

AMX will only stop or kill a task which is running, waiting or ready to execute. A task can be deleted if it is in any of these states or idle. Occasionally, a request to terminate a task will occur while that task is performing some operation which AMX deems to be critical. When this occurs, AMX allows the task to complete the critical operation and then forces the abnormal termination. AMX does this so that none of its private resources, such as message envelopes, are corrupted or lost.

When a task is stopped or killed, AMX makes sure that a task which was waiting for a message acknowledgement from the task being terminated, resumes and is not left blocked forever.

Deleting a task is a complex operation. AMX must make certain that the task is not deleted until all currently active transactions involving that task have been completed. For example, high priority Task A may be deleting medium priority Task B just as low priority Task C was interrupted in its attempt to trigger Task B. The task trigger must be allowed to complete in order that AMX task status not be compromised.

AMX meets these stringent requirements by requiring that you specify the priority at which a task is to be deleted. The task deletion priority must be below that of all other tasks with which it interacts but above any compute bound tasks which you may have. AMX drops the task to its deletion priority which forces all other tasks with which the task can interact to complete any outstanding operations before the deletion takes place. AMX then deletes the task.

Once a request to delete a task has been made, no new transactions involving the deleted task can occur. Any attempt to reference a task which has been marked for deletion will produce a *CJ_ERTKID* error indication since the deleted task no longer exists as far as you are concerned. The actual task deletion may be delayed by AMX until current transactions involving the task can be completed.

Termination Warning

When AMX stops, kills or deletes a task, it does not release any of the system resources which that task may have within its control. Any interval timers, semaphores, event groups, mailboxes, message exchanges, buffers or memory blocks which the task has reserved must be released by the task's Task Termination Procedure when it is called by AMX.

14.5 Task Suspend/Resume

Some operating systems permit a task to suspend any task, including itself. This feature is then used to implement the equivalent of the AMX task wait procedure `cjtkwait`.

The ability to arbitrarily suspend a task is one of the most abused privileges afforded by other operating systems. The AMX philosophy is to encourage well structured task designs using the wide range of task synchronization facilities offered by AMX in which there is rarely the need for an uncontrolled task suspension. In most cases, a modification to your system design can eliminate the apparent need to arbitrarily suspend a task.

To facilitate porting system designs from other operating systems to AMX, the procedures `cjtksuspend` and `cjtkresume` are provided.

Any Restart Procedure, task, ISP, Timer Procedure or Exit Procedure can suspend any task in the system except the AMX Kernel Task. The task remains suspended until some task, ISP or Timer Procedure calls `cjtkresume` to force resumption of that task.

If a task is blocked (waiting for a task trigger or in any AMX wait state) at the time it is suspended, the task will remain blocked (suspended) until a `cjtkresume` call attempts to let the task resume. If, while the task was suspended, the blocking conditions were removed, the task will resume (start) at the point at which it was blocked.

Note

Do not use `cjtksuspend` and `cjtkresume` for synchronization.

For task synchronization, see Chapter 3.6.
For ISP synchronization, see Chapter 4.4.

14.6 Message Exchange Tasks

A careful analysis of a large number of real-time multitasking applications shows that most include many message driven tasks, each with its own private message exchange. The tasks are often coded as illustrated in the following example. The task creates a message exchange and waits on it forever, processing each message that arrives at the exchange.

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

void CJ_CCPP usertask(void)
{
   CJ_ID    exchgid;                     /* Message exchange id      */
   struct cjxmsg msg;                    /* AMX message              */

                                         /* Create a message exchange*/
   if (cjmxcreate(&exchgid, "TKMX", 5, 10, 10, 0) == CJ_EROK) {

            for (;;) {                   /* Wait for message         */
                                         /* Wait at priority 0 forever*/
                      cjmxwait(exchgid, &msg, 0, 0);

                      userfunc(&msg);    /* Pass to real task        */

                      }
            }
   }
```

Recognizing this need, AMX offers the concept of a message exchange task which has a private message exchange tied directly to the task. The task and its private message exchange can be predefined in your AMX System Configuration Module. Using the Configuration Manager, you simply define the message queue depths for the task's message exchange as described in Chapter 15.5. The task and its private message exchange are then created automatically by AMX during the system startup.

A task and its private message exchange can also be created by you dynamically at run time. You use AMX procedure `cjtkbuild` or `cjtkcreate` to create the task and procedure `cjmxbuild` or `cjmxcreate` to create the message exchange. Then you call AMX procedure `cjtkmxinit` to tie the two together. The task must be created as illustrated in the example given in the description of `cjtkmxinit` in Section 3.

You must NOT trigger a message exchange task. AMX automatically triggers the task when the message exchange is attached to the task at startup or by your call to `cjtkmxinit`.

Standard AMX tasks do not receive any parameters when they are started at their entry point by AMX in response to a task trigger. However, a message exchange task receives an AMX message on the task's stack ready to be processed by the task.

How then are messages sent to such a task? Since the task's message exchange is a standard AMX message exchange, the message transmission process is exactly as described in Chapters 3.9 and 9. Any Restart Procedure, Exit Procedure, ISP, task or Timer Procedure can send a message to the task as long as it knows the task's message exchange id. A call to AMX procedure *cjtkmxid* will, given a task id, provide the message exchange id for that task if one exists.

Special care must be taken when terminating message exchange tasks. To stop such a task, you still use procedure *cjtkstop*. However, to kill or delete such a task, you must use the extended AMX task termination procedures *cjtkxkill* and *cjtkxdelete* respectively. These procedures ensure that when a task with a task exchange is killed or deleted, its message exchange is flushed and that all tasks, if any, waiting for message acknowledgement are allowed to resume with an answer-back status of *CJ_WRTKFLUSH*.

Note

You must configure your AMX system to include enough message exchanges to allow one exchange to be allocated for each message exchange task which you need.

Example See the description of procedure *cjtkmxinit* in Section 3.

This page left blank intentionally.

15. AMX System Configuration

15.1 System Configuration Module

The AMX System Configuration Module defines the characteristics of your AMX system. The AMX Configuration Builder, described in Chapter 15.2, will create this module for you.

The System Configuration Module includes the following components.

The **User Parameter Table** provides AMX with the following information pertinent to the system you are creating.

Maximum number of tasks	Number of message envelopes
Maximum number of timers	Message size
Maximum number of semaphores	Kernel and Interrupt Stack sizes
Maximum number of event groups	Clock frequency
Maximum number of mailboxes	Time/Date Scheduling Procedure
Maximum number of message exchanges	
Maximum number of buffer pools	
Maximum number of memory pools	

The **Restart Procedure List** defines all of your Restart Procedures and their order of execution. The **Exit Procedure List** defines all of your Exit Procedures and their order of execution.

If you wish, you can predefine one or more of any of the following components of your system. The object definition and memory storage, if any, required by AMX for the component are provided in the System Configuration Module.

Tasks (including message exchange tasks)	Mailboxes
Timers	Message exchanges
Semaphores (resource or counting)	Buffer pools
Event groups	Memory pools

If you predefine any of these components, AMX will automatically create them when your system starts. The id assigned to each of the predefined components will be stored in unique **id variables** which are in the System Configuration Module.

If you predefine any tasks, the **task stacks** for these tasks will be in the System Configuration Module.

If you predefine any buffer pools, the **buffers** for these buffer pools will be in the System Configuration Module.

If you predefine any **message exchange tasks**, the tasks and their predefined message exchanges will be defined in the System Configuration Module. The message exchange tasks will be automatically created and started by AMX when your system is launched.

15.2 System Configuration Builder

The AMX Configuration Builder is a software generation tool which can be used to create your AMX System Configuration Module. The Builder helps to reduce total system implementation time by eliminating the manual generation process by which your System Configuration Module would otherwise have to be produced. The Builder consists of two components: the Configuration Manager and the Configuration Generator. The Configuration Manager is an interactive utility which allows you to define and edit all of the parameters which must be included in your System Configuration Module to use AMX and its Managers.

The configuration process is illustrated in the block diagram of Figure 15.2-1.

The Configuration Manager lets you define your system to meet your needs. It produces a text file called a User Parameter File. This file contains a cryptic representation of your description of your system.

The Configuration Manager is also able to read a User Parameter File allowing you to use the Configuration Manager to edit the content of a previously defined system description.

For convenience, the Configuration Manager has the ability to directly invoke its own copy of the Configuration Generator. The Configuration Generator reads your User Parameter File and uses the information in it to produce a source file called the System Configuration Module and a text file called the System Documentation Module.

The Configuration Generator uses a file called the System Configuration Template as a model for your System Configuration Module. This template file is merged with the information in your User Parameter File to produce your System Configuration Module.

The Configuration Generator also merges a file called the System Documentation Template with the information in your User Parameter File to produce your System Documentation Module, a text file summarizing the characteristics of your AMX configuration.

The C language System Configuration Module must be compiled as described in the compiler specific chapter of the AMX Tool Guide for inclusion in your AMX system.

If you are not using one of the C compilers supported by KADAK, you may still be able to use the AMX Configuration Builder by altering the System Configuration Template File as described in Appendix C.

The Configuration Generator is also available as a separate, stand alone DOS utility. This utility program can be used within your make files to generate and then compile your System Configuration Module. Instructions for doing so are provided in the AMX Tool Guide.

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.

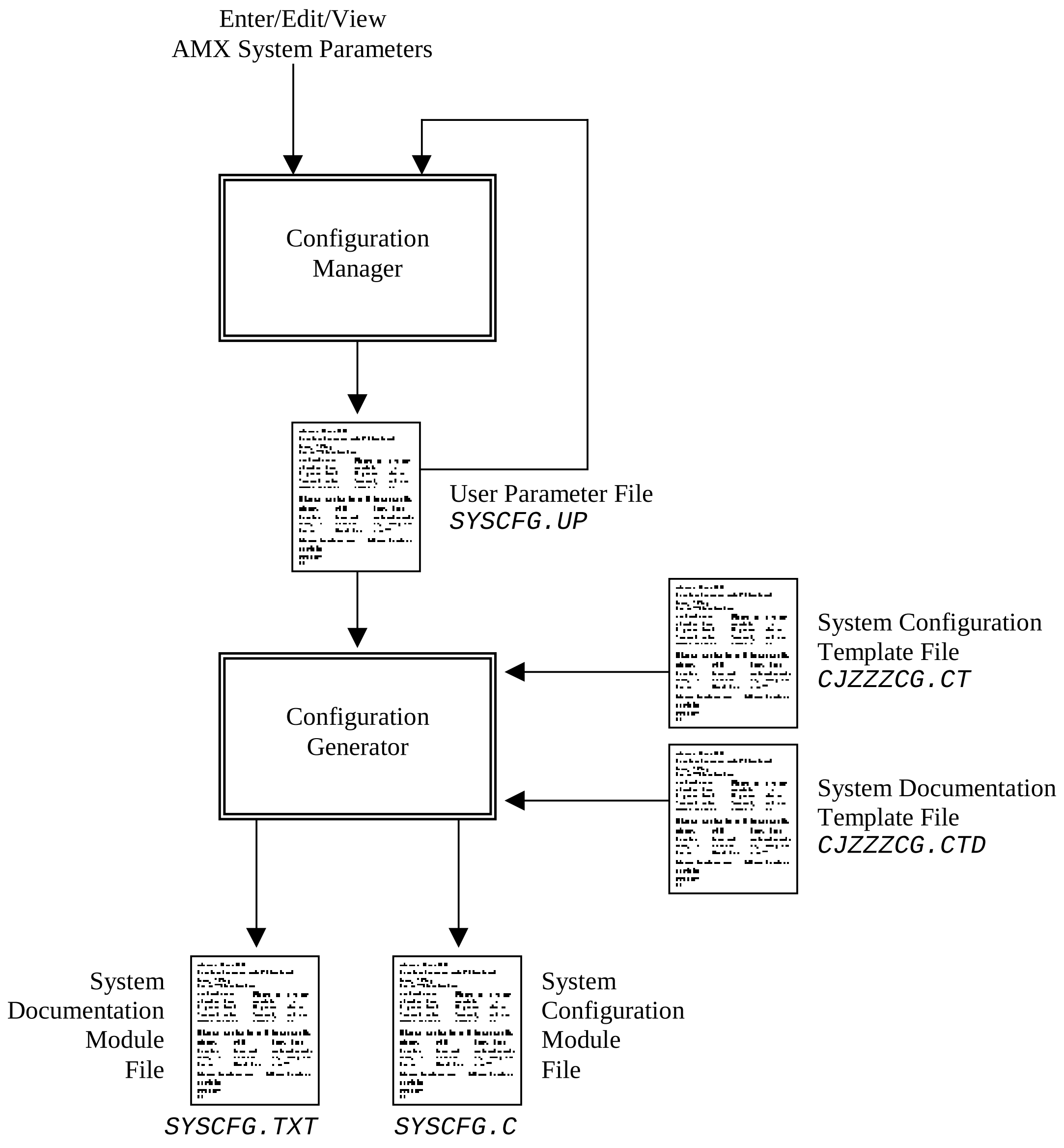


Figure 15.2-1 AMX Configuration Building Process

15.3 Using the Builder

Starting the Builder

The AMX Configuration Builder will operate on a PC or compatible running the Microsoft® Windows® operating system.

The Builder is delivered with the following files.

File	Purpose
CJZZZCM .EXE	AMX Configuration Manager (utility program)
CJZZZCM .CNT	AMX Configuration Manager Help Content File
CJZZZCM .HLP	AMX Configuration Manager Help File
CJZZZCG .EXE	AMX Configuration Generator (utility program)
CJZZZCG .CT	System Configuration Template File
CJZZZCG .CTD	System Documentation Template File

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

To open an existing User Parameter File, say *SYSCFG.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 system configuration, select Save from the File menu. The Configuration Manager will rename your original User Parameter File to be *SYSCFG.BAK* and create an updated version of the file called *SYSCFG.UP*.

To direct the Configuration Manager to use its Configuration Generator utility to produce an updated copy of your System Configuration Module, say *SYSCFG.C*, select Generate... from the File menu.

The Configuration Manager can also be directed to use its Configuration Generator utility to produce a System Documentation Module, say *SYSCFG.TXT*, a text file summarizing the characteristics of your AMX configuration. Select Document... from the File menu.

Note

AMX configuration constants (minimums, maximums, limits, etc.) are provided in the processor specific AMX Target Guide.

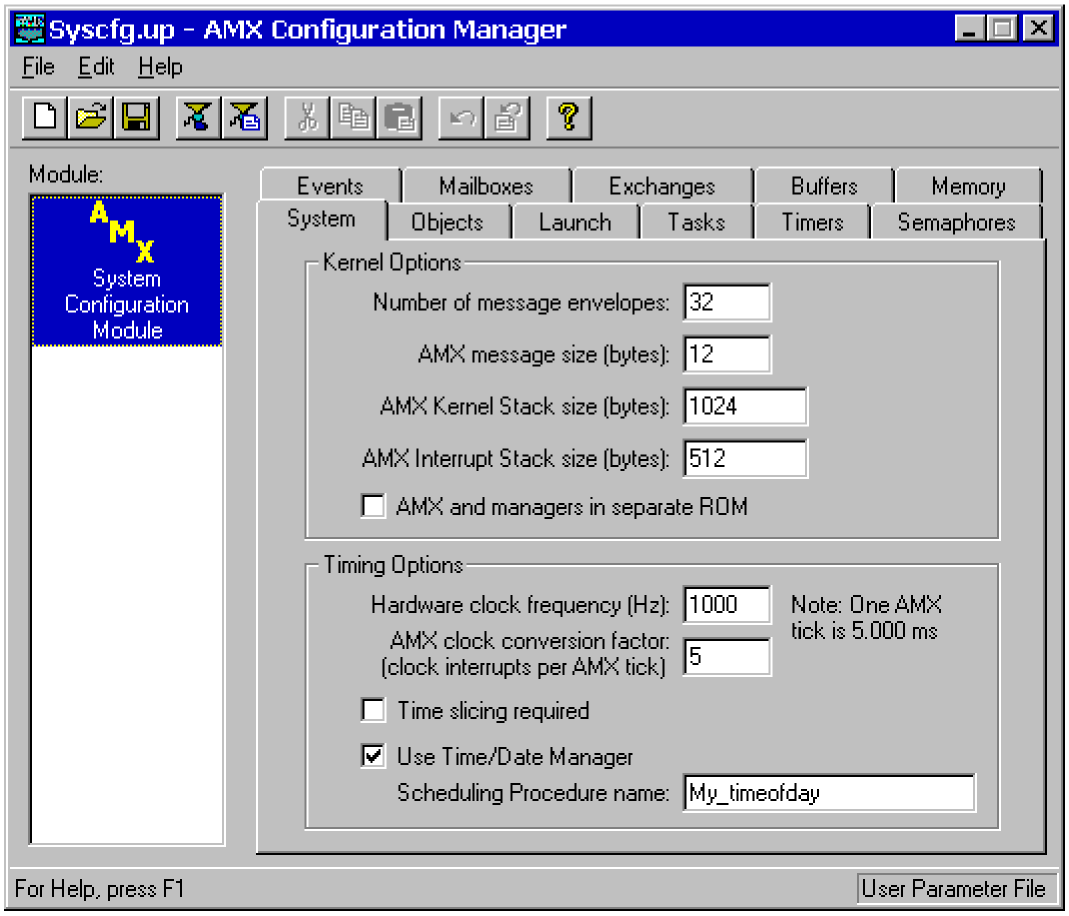
Screen Layout

Figure 15.3-1 illustrates the Configuration Manager's screen layout. The title bar identifies the User 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 System Configuration Module selector must be active to generate the System Configuration Module and its related documentation file.

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



Kernel Options

Number of message envelopes: 32

AMX message size (bytes): 12

AMX Kernel Stack size (bytes): 1024

AMX Interrupt Stack size (bytes): 512

☐ AMX and managers in separate ROM

Timing Options

Hardware clock frequency (Hz): 1000

AMX clock conversion factor: (clock interrupts per AMX tick) 5

Note: One AMX tick is 5.000 ms

☐ Time slicing required

☑ Use Time/Date Manager

Scheduling Procedure name: My_timeofday

For Help, press F1

User Parameter File

Figure 15.3-1 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 User Parameter File. It also provides the Exit command.

When the System Configuration Module selector icon is the currently active selector, the Generate... command on the File menu can be used to generate your System Configuration Module and the File, Document... command can be used to generate your System Documentation Module. The paths to the template files required by the generator to create these products 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 System Configuration Module selector icon is the currently active selector, the System 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 system configuration parameters can be declared. For instance, if you select the Tasks tab, the Configuration Manager will present a task definition window (property page) containing all of the parameters you must provide to completely define a task.

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 User 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 AMX Objects

Separate property pages are provided to allow your definition of one or more AMX objects such as tasks and timers which will be prebuilt by AMX when AMX is launched. The maximum allowed quantities of each type of AMX object are defined by you on the Objects property page.

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 is a counter showing the number of objects in the list and the allowable maximum number of objects as defined on the Objects property page.

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 named ---- will appear at the bottom of the list and will be opened ready for editing. When you enter a valid tag for the object, the tag will replace the name ---- in the object list.

To edit an existing object's definition, double click on the object's name 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 name 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 name 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.

Tasks are ordered in their object list by task priority. The highest priority task resides at the top of the list. Tasks cannot be rearranged in their list by dragging the object. To do so, change the task's priority.

15.4 System Parameter Definition

The System Parameter window allows you to define the general operating parameters of your AMX system. The layout of the window is shown in Figure 15.3-1 in Chapter 15.3.

Kernel Options

AMX Message Envelopes

AMX passes a message to a mailbox or message exchange in a message envelope. The application parameters which form the message are copied into the envelope and added to the mailbox or message exchange.

The number of message envelopes required by your application must be defined. The upper limit of the maximum number of envelopes is target dependent. The minimum number of envelopes is `CJ_MINKG` bytes. For each of these message envelopes, the Builder will allocate storage for private AMX use.

Unfortunately, no hard and fast rule can be provided to determine the number of message envelopes needed. For simple systems with little message queuing and few tasks executing concurrently, the number of required message envelopes can range from one to twice the number of tasks in the system. More complex systems with significant queuing requirements may require many more envelopes.

AMX Message Size

The maximum size of the message to be passed in a message envelope must also be defined by you. If you enter a size of 0, the Manager will report the minimum allowed. The minimum buffer size is `CJ_MINMSZ` bytes. The message size must be a multiple of 4.

If you change the message size from its default value, you must edit the definition of `CJ_MAXMSZ` in AMX header file `CJZZZAPP.H` to reflect your new message size. Be sure to recompile all application modules which manipulate messages.

AMX Kernel Stack Size

The AMX Kernel Task requires a minimum Kernel Stack size which depends on your target processor. If you enter a size of 0, the Manager will report the minimum allowed. The minimum Kernel Stack size is `CJ_MINKS` bytes. The stack size must be a multiple of 4 bytes.

In addition to this minimum, you must allocate sufficient stack to satisfy the worst case requirements of all application Restart Procedures and Timer Procedures. If you have any nonconforming ISPs, you must increase the Kernel Stack size to meet the interrupt overhead expected.

AMX Interrupt Stack Size

The AMX Interrupt Supervisor requires a minimum Interrupt Stack size which depends on your target processor. If you enter a size of 0, the Manager will report the minimum allowed. The minimum Interrupt Stack size is *CJ_MINIS* bytes. The stack size must be a multiple of 4 bytes.

In addition to this minimum, you must allocate sufficient stack to satisfy the application ISP with the greatest stack usage. If nested interrupts are possible, then the worst case interrupt nesting must be analyzed and additional interrupt stack allocated.

AMX and Managers in Separate ROM

AMX and its managers can be installed in a private ROM and accessed by your application via the AMX ROM access module. If you are using AMX this way, check this box.

If you are linking your application with AMX and its managers, leave this box unchecked even if you are going to burn your linked AMX system into a ROM.

Timing Options

Hardware Clock Frequency

This parameter defines the frequency of the AMX hardware clock in hertz. It is used by AMX to convert milliseconds to equivalent AMX system ticks. If your hardware clock frequency is not integral, round the clock frequency to the nearest non-zero integer.

AMX Clock Conversion Factor

The AMX system tick is measured in multiples of hardware clock interrupts. This parameter specifies the number of hardware clock interrupts required for each system tick. For example, a 100 Hz clock will generate interrupts at 10 millisecond intervals. If a 50 ms system tick is required, then this parameter must be set to 5.

Time Slicing Required

If you wish to allow tasks which share a priority level to be time sliced, check this box. Leave this box unchecked if time slicing of tasks is not required.

If time slicing has been enabled and a task has been defined to have a non-zero time slice, you will not be able to remove the check from this box. You must first set the time slice value for all defined tasks to 0. Then you will be able to remove the check from this box, thereby disabling time slicing.

Use Time/Date Manager

If you wish to include the Time/Date Manager, check this box. Otherwise leave the box unchecked. Be sure to increase the maximum number of timers in your system by one to account for the AMX Time/Date timer.

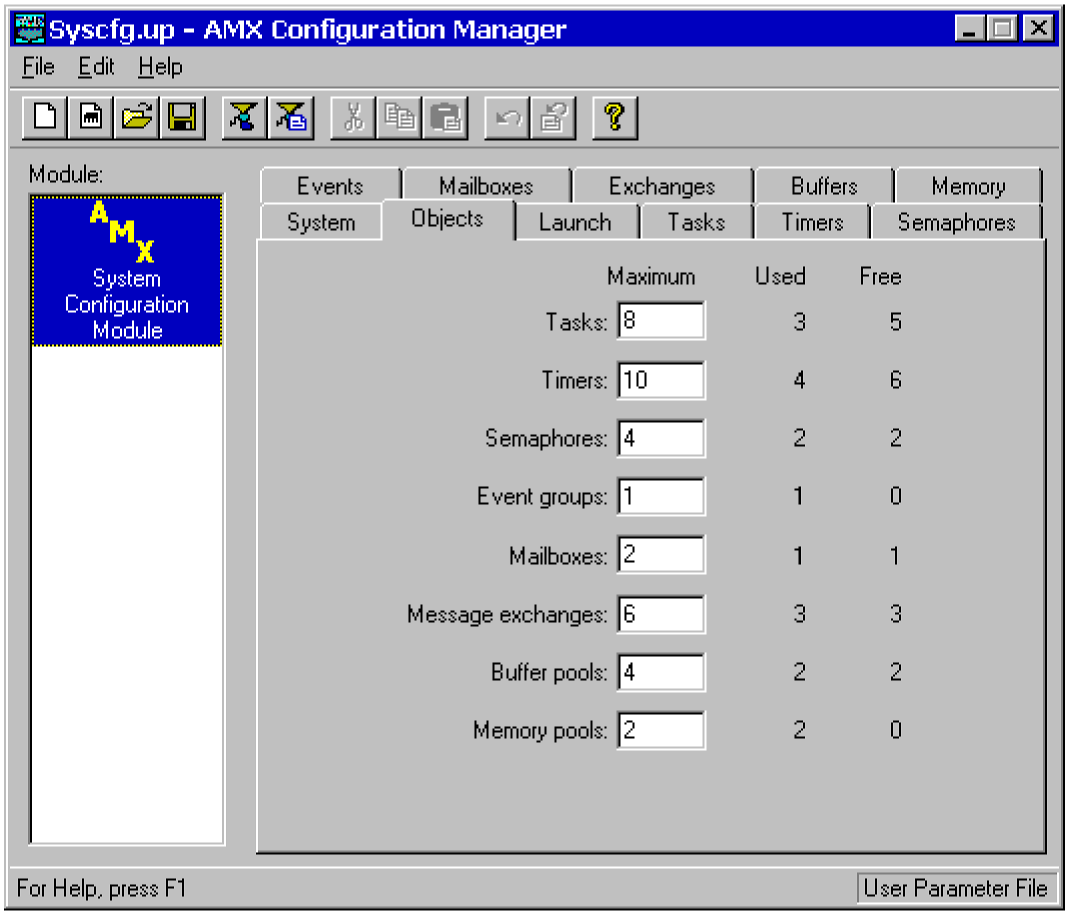
Scheduling Procedure Name

If you use the Time/Date Manager, then you may wish to define a Time/Date Scheduling Procedure (see Chapter 5.5). This parameter defines that procedure name. If no procedure is required, leave this field empty (blank).

This field is ignored if the Use Time/Date Manager check box is unchecked.

15.5 AMX Object Allocation

The AMX Object Allocation window allows you to define the number of private AMX objects required for each of the optional AMX managers to be included in your system. The layout of the window is shown below.



	Maximum	Used	Free
Tasks:	8	3	5
Timers:	10	4	6
Semaphores:	4	2	2
Event groups:	1	1	0
Mailboxes:	2	1	1
Message exchanges:	6	3	3
Buffer pools:	4	2	2
Memory pools:	2	2	0

Maximum Number of Tasks

This parameter defines the maximum number of application tasks which your system can support concurrently. The value for this parameter should be the number of predefined tasks (see Chapter 15.7) plus the maximum number of tasks that you will dynamically create using `cjtkbuild` or `cjtkcreate`. You do not have to account for the AMX Kernel Task.

Maximum Number of Timers

This parameter defines the maximum number of timers which your system can support concurrently. If you do not require any timers, set this parameter to 0. Otherwise, this parameter should be set to the number of predefined timers (see Timer Definition Window) plus the maximum number of timers that you dynamically create using `cjtmbuild` or `cjtmcreate`.

Maximum Number of Semaphores

This parameter defines the maximum number of resource and counting semaphores which your system can support concurrently. If you do not require any semaphores, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined semaphores (see Semaphore Definition Window) plus the maximum number of resource or counting semaphores that you will dynamically create using *`cjrmbuild`*, *`cjrmcreate`*, *`cjsmbuild`* or *`cjsmcreate`*.

Maximum Number of Event Groups

This parameter defines the maximum number of event groups which your system can support concurrently. If you do not require any event groups, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined event groups (see Event Group Definition Window) plus the maximum number of event groups that you will dynamically create using *`cjevbuild`* or *`cjevcreate`*.

Maximum Number of Mailboxes

This parameter defines the maximum number of mailboxes which your system can support concurrently. If you do not require any mailboxes, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined mailboxes (see Mailbox Definition Window) plus the maximum number of mailboxes that you will dynamically create using *`cjmbbuild`* or *`cjmbcreate`*.

Maximum Number of Message Exchanges

This parameter defines the maximum number of message exchanges which your system can support concurrently. If you do not require any message exchanges, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined message exchanges (see Message Exchange Definition Window) plus the maximum number of message exchanges that you will dynamically create using *`cjmxbuild`* or *`cjmxcreate`*.

You must also account for the private message exchanges which will be needed for the message exchange tasks, if any, that you intend to predefine or dynamically create.

Maximum Number of Buffer Pools

This parameter defines the maximum number of buffer pools which your system can support concurrently. If you do not require any buffer pools, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined buffer pools (see Buffer Pool Definition Window) plus the maximum number of buffer pools that you will dynamically create using *`cjbmbuild`* or *`cjbmcreate`*.

Maximum Number of Memory Pools

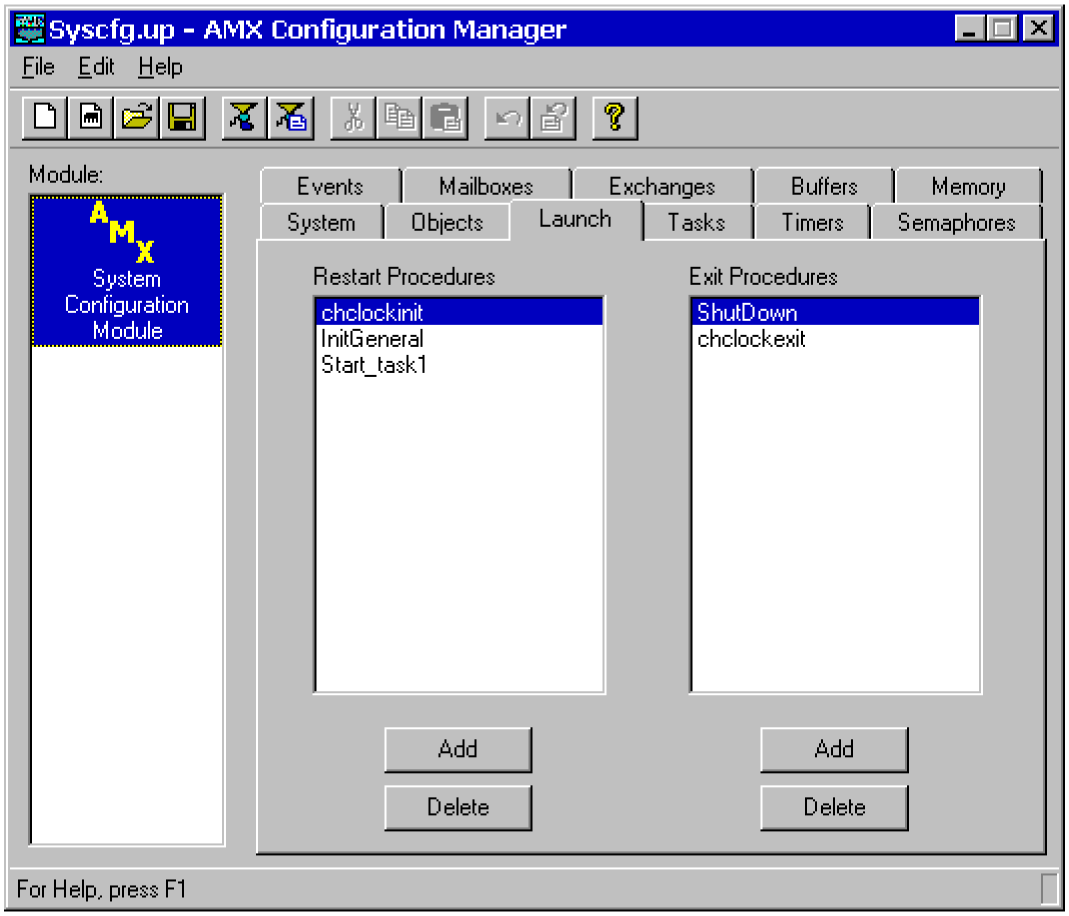
This parameter defines the maximum number of memory pools which your system can support concurrently. If you do not require any memory pools, set this parameter to 0. Otherwise, the parameter should be set to the number of predefined memory pools (see Memory Pool Definition Window) plus the maximum number of memory pools that you will dynamically create using *`cjmmbuild`* or *`cjmmcreate`*.

15.6 Restart/Exit Procedure Definition

The Launch/Shutdown window displays all of your application Restart and Exit Procedures which will be called by AMX at system startup and shutdown. The layout of the window is shown below.

The Restart Procedure List is used to define all of your application Restart Procedures which will be called by AMX at system startup. The Restart Procedures will be called by AMX in the order in which they appear in the list.

The Exit Procedure List is used to define all of your application Exit Procedures which will be called by AMX at system shutdown. The Exit Procedures will be called by AMX in the order in which they appear in the list.



Restart Procedures

chclockinit
InitGeneral
Start_task1

Exit Procedures

ShutDown
chclockexit

Add, Edit and Delete Restart and Exit Procedures

To add a new procedure, click on the `Add` button below the list. A new procedure named `---New---` will appear at the bottom of the list. Click on the name `---New---` and it will be opened ready for editing. Enter the name of your procedure.

To edit an existing procedure's name, double click on the name in the object list. The name will be opened ready for editing.

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

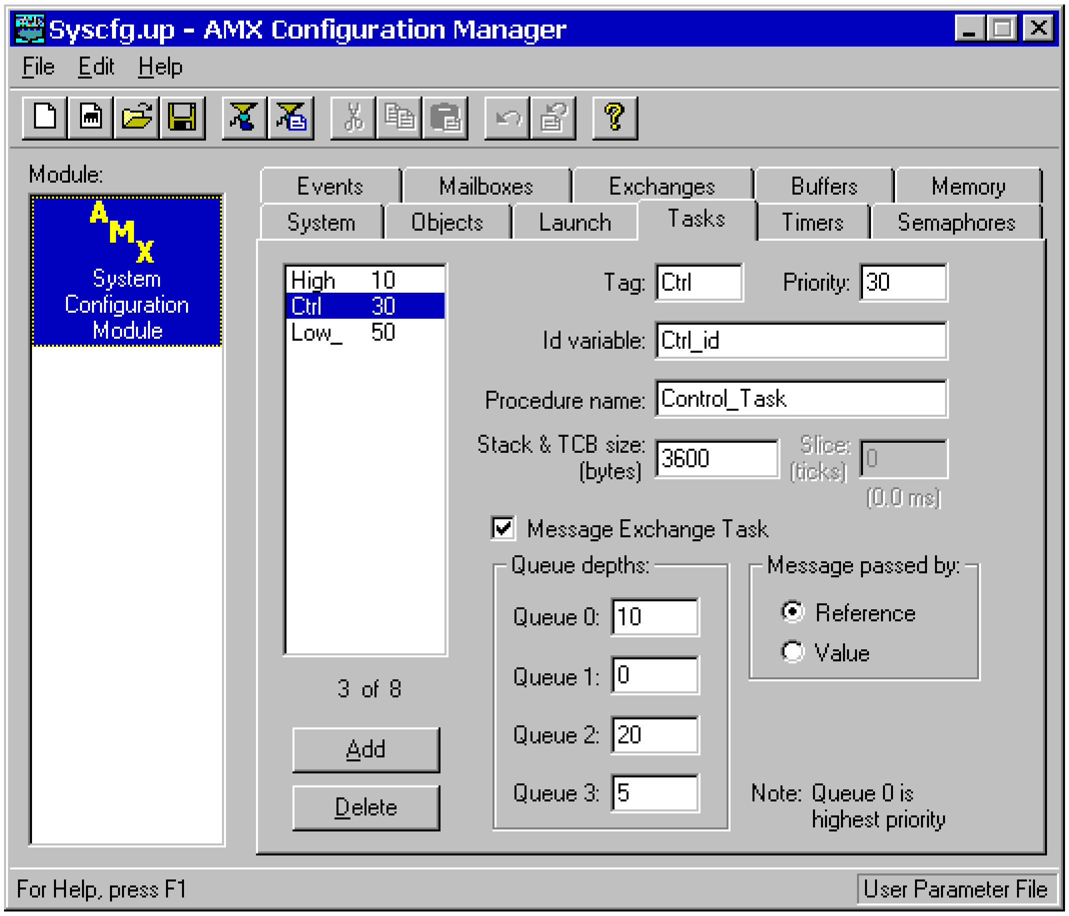
The procedure names in the object list can be rearranged by dragging a procedure's name to the desired position in the list. You cannot drag a procedure directly to the end of the list. To do so, first drag the procedure name to precede the last name on the list. Then drag the last name on the list to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

Note

The Restart and Exit Procedures for all AMX Managers will be automatically provided by the Configuration Manager. You MUST NOT repeat them in your list of application Restart or Exit Procedures.

15.7 Task Definition

The Task Definition window allows you to define the tasks to be automatically created by AMX at system startup. You do not have to predefine all of your tasks in this manner; you may also create tasks dynamically using `cjtkbuild` or `cjtkcreate`. The layout of the window is shown below.


Syscfg.up - AMX Configuration Manager
File Edit Help
Module:
AMX System Configuration Module
Events Mailboxes Exchanges Buffers Memory
System Objects Launch Tasks Timers Semaphores
Add
Delete
For Help, press F1
User Parameter File

High	10
Ctrl	30
Low_	50

3 of 8

Tag: Ctrl Priority: 30
Id variable: Ctrl_id
Procedure name: Control_Task
Stack & TCB size: (bytes) 3600 Slice: (ticks) 0 (0.0 ms)
[✓] Message Exchange Task
Queue depths: Queue 0: 10 Queue 1: 0 Queue 2: 20 Queue 3: 5
Message passed by: [✓ Reference] Value
Note: Queue 0 is highest priority

Task Tag

Each task can have a unique 4-character task tag. This parameter defines that tag. Although AMX does not restrict the content of the task tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Priority

This parameter defines the execution priority of the task. Application task priorities range from 1 (highest) to 127 (lowest). More than one task can be given the same task priority. Unless the task is time sliced, AMX will assign relative task priorities according to the order in which the task definitions appear on the screen.

Id Variable

This parameter defines the name of a public variable of type *`CJ_ID`* in the System Configuration Module in which AMX will save the task id of the task.

Task Procedure Name

This parameter defines the name of the procedure to be called when AMX starts the task.

Stack & TCB Size

Each application task requires a block of RAM storage for use as a stack and Task Control Block (TCB). The block size depends on your target processor. If you enter a size of 0, the Manager will report the minimum allowed. The minimum block size is *`CJ_MINTKS`* bytes. In addition to this minimum, you must allocate sufficient stack to satisfy the worst case requirements of the task. The stack size parameter must be a multiple of 4 bytes.

If your task installs its own task trap handlers (see Chapter 4.5), increase your task's stack by *`sizeof(struct cjxregs)`* bytes.

If you have any nonconforming ISPs, you must increase the stack size of all tasks to meet the needs of these ISPs.

Time Slice

If the task is to be time sliced, you may specify the time slice interval in the task definition. If time slicing is not required, set this parameter to 0.

This field will be disabled unless you have enabled the AMX time slice option by checking the Time Slicing Required box on the System Parameter Window.

Time slicing is only effective if the task will be compute bound in competition with other tasks of equal priority.

The task's time slice may also be altered dynamically by calling *`cjtmslice`* when your system is running.

Message Exchange Task

AMX allows creation of a message exchange task (see Chapter 14.6) for which a private message exchange is allocated. To declare a task of this type, check this box and fill in the required queue depths for the task's private message exchange. Otherwise leave the box unchecked.

Queue Depths

These four parameters define the maximum number of message envelopes which can reside in each of the four message queues of the task's private message exchange. Queue 0 is the highest priority queue; queue 3 is the lowest priority.

Depths may range from 0 to 32767. The maximum depth of a queue does not affect AMX memory requirements. If a particular message queue is to be unused, set its depth to zero. At least one message queue depth must be non-zero if the task is to be defined as a message exchange task.

The message exchange's tag is the same as the task's tag. AMX stores the id of the task's message exchange in an id variable with name *`YOURTASKID_mx`*, where *`YOURTASKID`* is the name you entered for the task's id variable.

Message Passed By

This option defines whether the message exchange task will receive its messages by reference or by value.

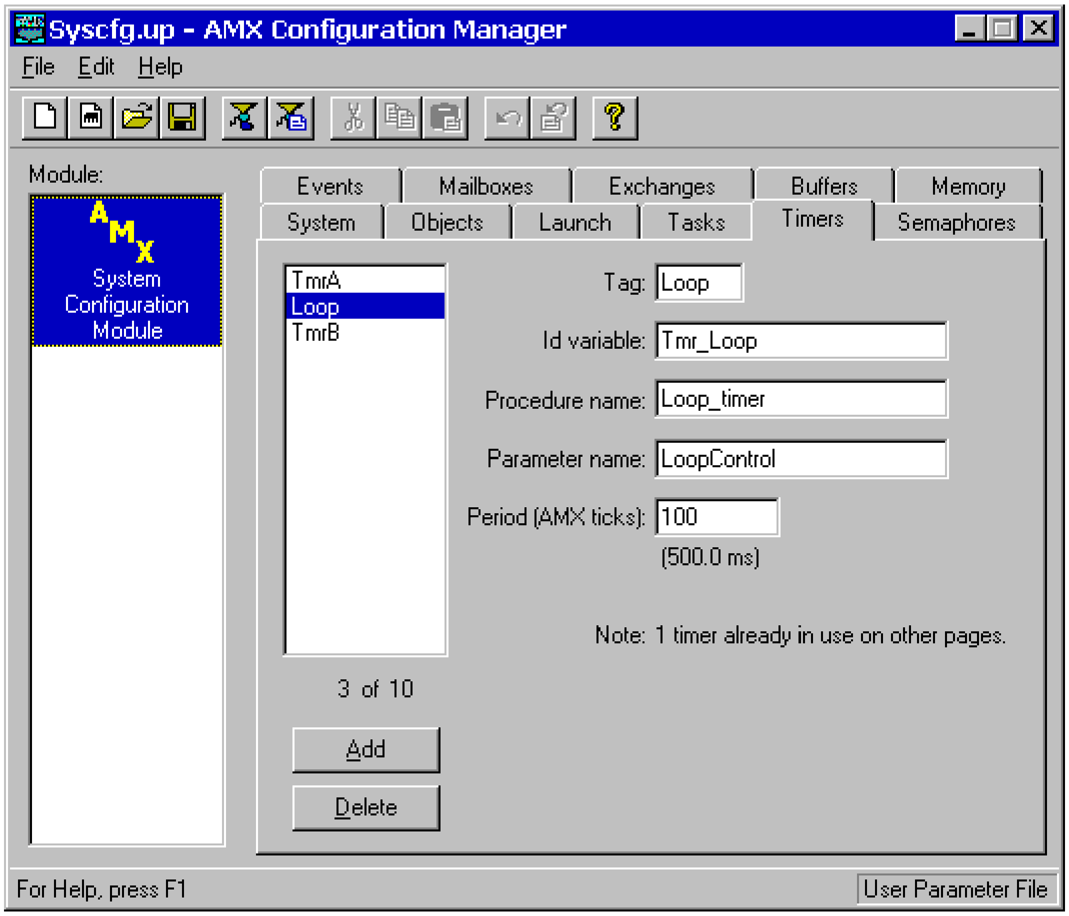
When a message is passed by reference, the task receives a pointer to a copy of the AMX message stored on the task's stack.

When a message is passed by value, the task receives the AMX message as a *`cjxmsg`* structure passed by value according to the parameter passing conventions of the C compiler being used.

15.8 AMX Object Definitions

Timer Definition

The Timer Definition window allows you to define the timers to be automatically created by AMX at system startup. You do not have to predefine all of your timers in this manner; you may also create timers dynamically using `cjtmbuild` or `cjtmcreate`. Note that AMX does not automatically start timers defined this way. Your application must call `cjtmwrite` to actually start the timer. The layout of the window is shown below.



TmrA
Loop
TmrB

3 of 10

Tag: Loop

Id variable: Tmr_Loop

Procedure name: Loop_timer

Parameter name: LoopControl

Period (AMX ticks): 100
(500.0 ms)

Note: 1 timer already in use on other pages.

Tag

Each timer can have a unique 4-character timer tag. This parameter defines that tag. Although AMX does not restrict the content of the timer tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *CJ_ID* in the System Configuration Module in which AMX will save the timer id of the timer.

Timer Procedure Name

This parameter defines the name of the Timer Procedure to be called whenever the timer expires.

Timer Parameter Name

This parameter defines the name of a public pointer variable whose address will be passed as a parameter to the Timer Procedure. If your timer does not require such a parameter, leave this field empty (blank).

If you enter the name *tmrparam*, the pointer variable *tmparam* will be declared as follows in the System Configuration Module:

```
extern void * tmrparam;
```

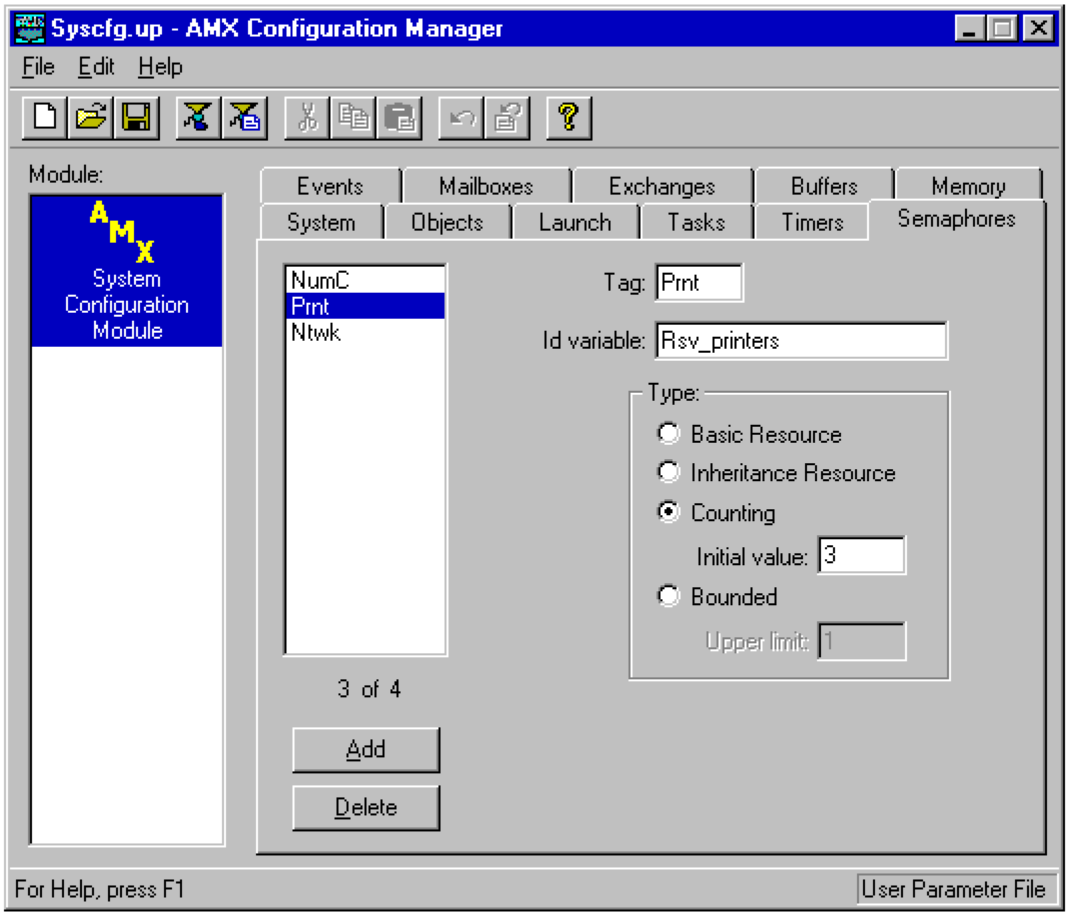
Your Timer Procedure will then receive *&tmrparam* as its parameter.

Period

If the timer is a periodic timer, this parameter defines its period in multiples of AMX system ticks. If the timer is a one-shot timer, this parameter must be set to 0.

Semaphore Definition

The Semaphore Definition window allows you to define the semaphores to be automatically created by AMX at system startup. You do not have to predefine all of your semaphores in this manner; you may also dynamically create resource semaphores using `cjrmbuild`, `cjrmcreate` or `cjrmcreatex` and counting semaphores using `cjsmbuild` or `cjsmcreate`. The layout of the window is shown below.



NumC
Prnt
Ntwk

3 of 4

Tag: Prnt

Id variable: Rsv_printers

Type:
- Basic Resource
- Inheritance Resource
- [✓ Counting]
  - Initial value: 3
- Bounded
  - Upper limit: 1

Tag

Each semaphore can have a unique 4-character semaphore tag. This parameter defines that tag. Although AMX does not restrict the content of the semaphore tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *CJ_ID* in the System Configuration Module in which AMX will save the semaphore id of the semaphore.

Type

This option field defines the type of semaphore to be created, Basic Resource, Inheritance Resource, Counting or Bounded.

Initial Value

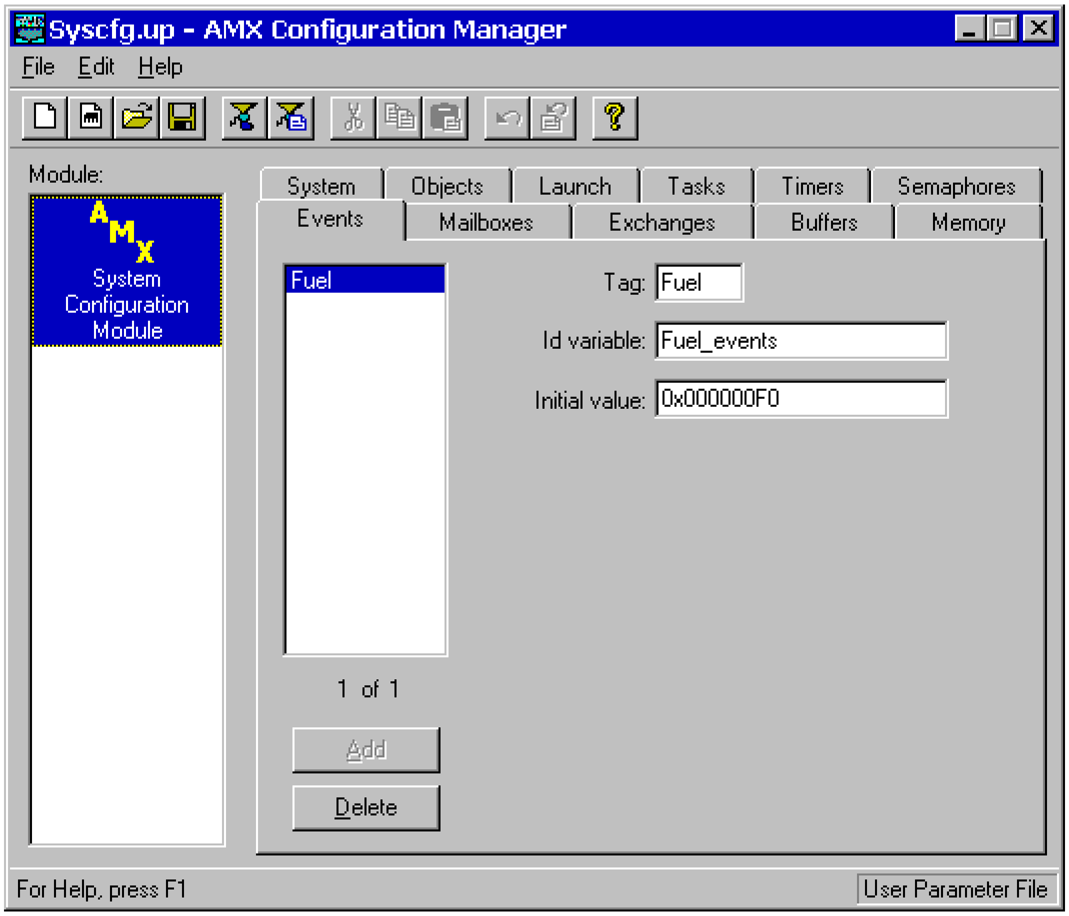
Counting semaphores may be given an initial value between 0 and 16383. This field is ignored for all semaphore types other than counting semaphores.

Upper Limit

Bounded semaphores may be given an upper limit between 1 and 16383. When a bounded semaphore is created, it will be assigned an initial value of 0. This field is ignored for all semaphore types other than bounded semaphores.

Event Group Definition

The Event Group Definition window allows you to define the event groups to be automatically created by AMX at system startup. You do not have to predefine all of your event groups in this manner; you may also create event groups dynamically using `cjevbuild` or `cjevcreate`. The layout of the window is shown below.



Tag: Fuel
Id variable: Fuel_events
Initial value: 0x000000F0

Tag

Each event group can have a unique 4-character event group tag. This parameter defines that tag. Although AMX does not restrict the content of the event group tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

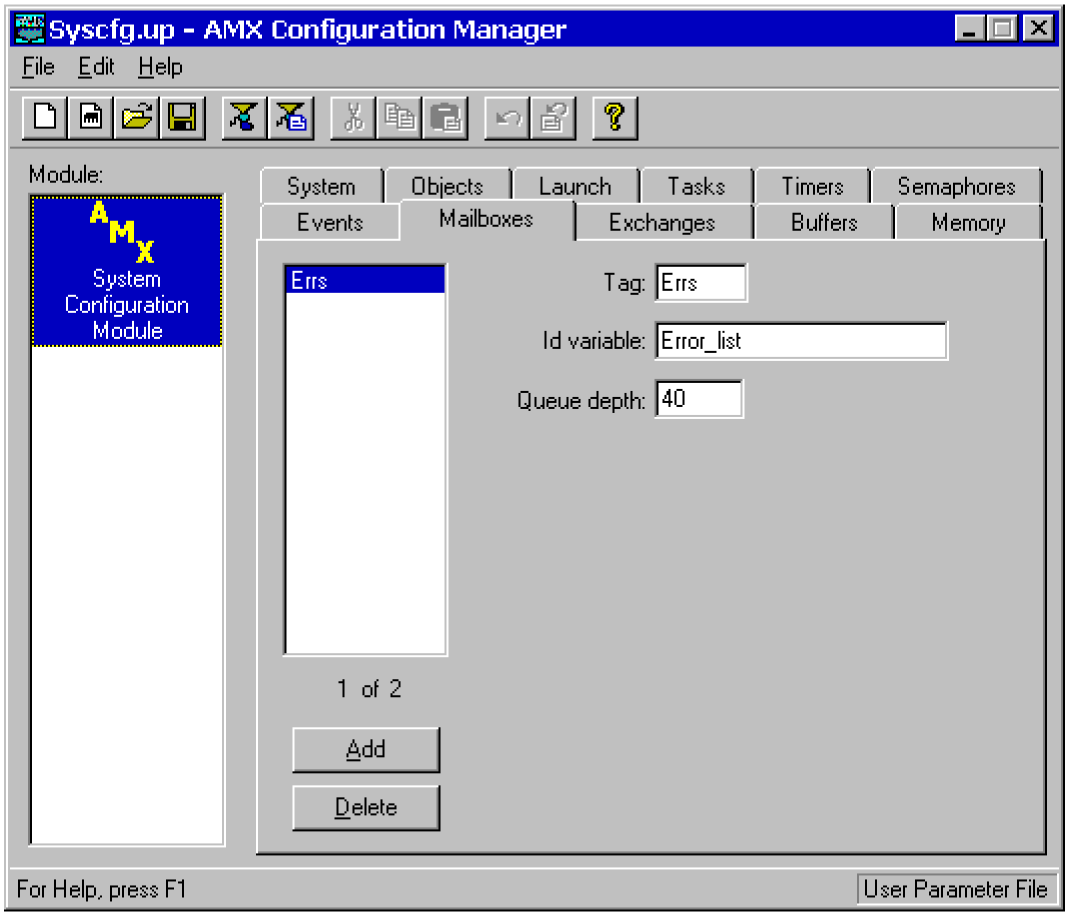
This parameter defines the name of a public variable of type *`CJ_ID`* in the System Configuration Module in which AMX will save the event group id of the event group.

Initial Value

This parameter defines the initial value of the 16 or 32 event flags in the event group. This parameter must be entered as 0 or as an unsigned hexadecimal number of the form *`0xdddddddd`*.

Mailbox Definition

The Mailbox Definition window allows you to define the mailboxes to be automatically created by AMX at system startup. You do not have to predefine all of your mailboxes in this manner; you may also create mailboxes dynamically using `cjmbbuild` or `cjmbcreate`. The layout of the window is shown below.



Errs

Tag: Errs

Id variable: Error_list

Queue depth: 40

1 of 2

Tag

Each mailbox can have a unique 4-character mailbox tag. This parameter defines that tag. Although AMX does not restrict the content of the mailbox tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

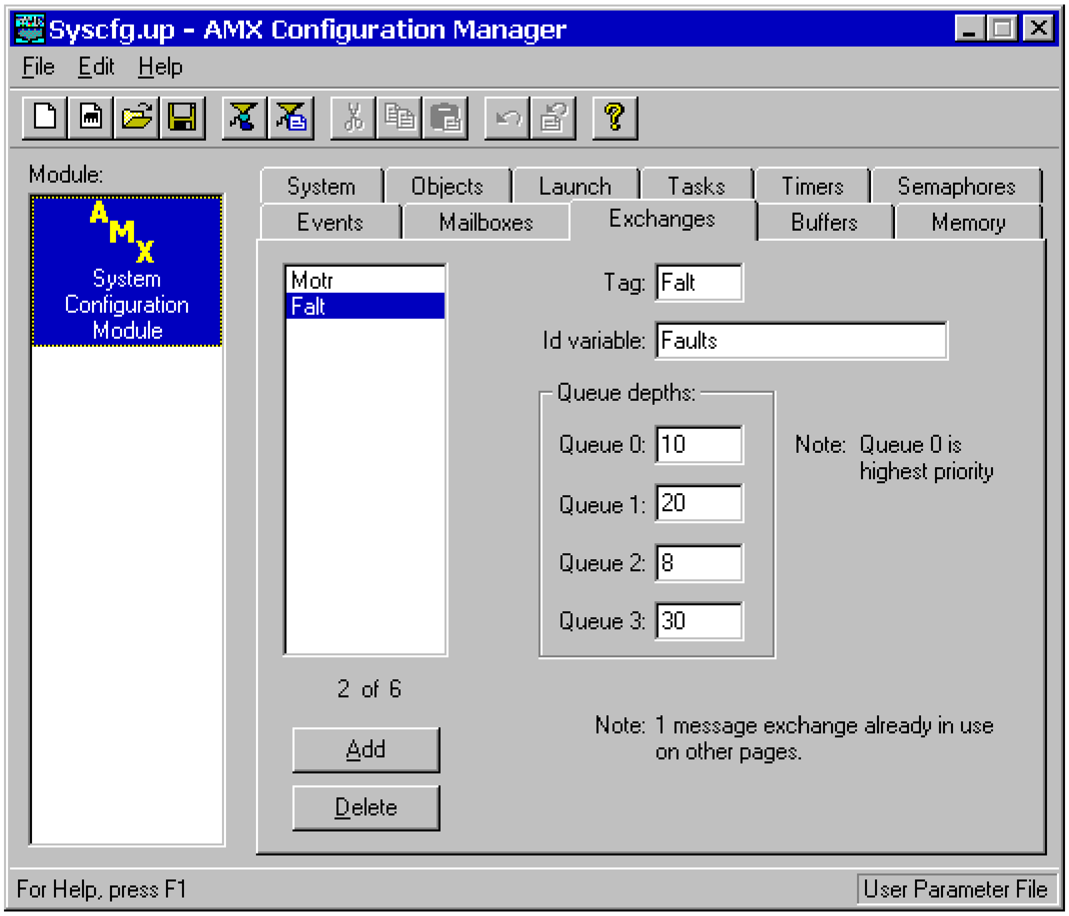
This parameter defines the name of a public variable of type *CJ_ID* in the System Configuration Module in which AMX will save the mailbox id of the mailbox.

Queue Depth

This parameter defines the maximum number of message envelopes which can reside in the mailbox message queue. Depths may range from 1 to 32767. The maximum depth for a mailbox does not affect AMX memory requirements.

Message Exchange Definition

The Message Exchange Definition window allows you to define the message exchanges to be automatically created by AMX at system startup. You do not have to predefine all of your message exchanges in this manner; you may also create message exchanges dynamically using `cjmxbuild` or `cjmxcreate`. The layout of the window is shown below.



Motr
Falt

Tag: Falt

Id variable: Faults

Queue depths:

Queue 0: 10

Queue 1: 20

Queue 2: 8

Queue 3: 30

Note: Queue 0 is highest priority

2 of 6

Note: 1 message exchange already in use on other pages.

Tag

Each message exchange can have a unique 4-character message exchange tag. This parameter defines that tag. Although AMX does not restrict the content of the message exchange tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *CJ_ID* in the System Configuration Module in which AMX will save the message exchange id of the message exchange.

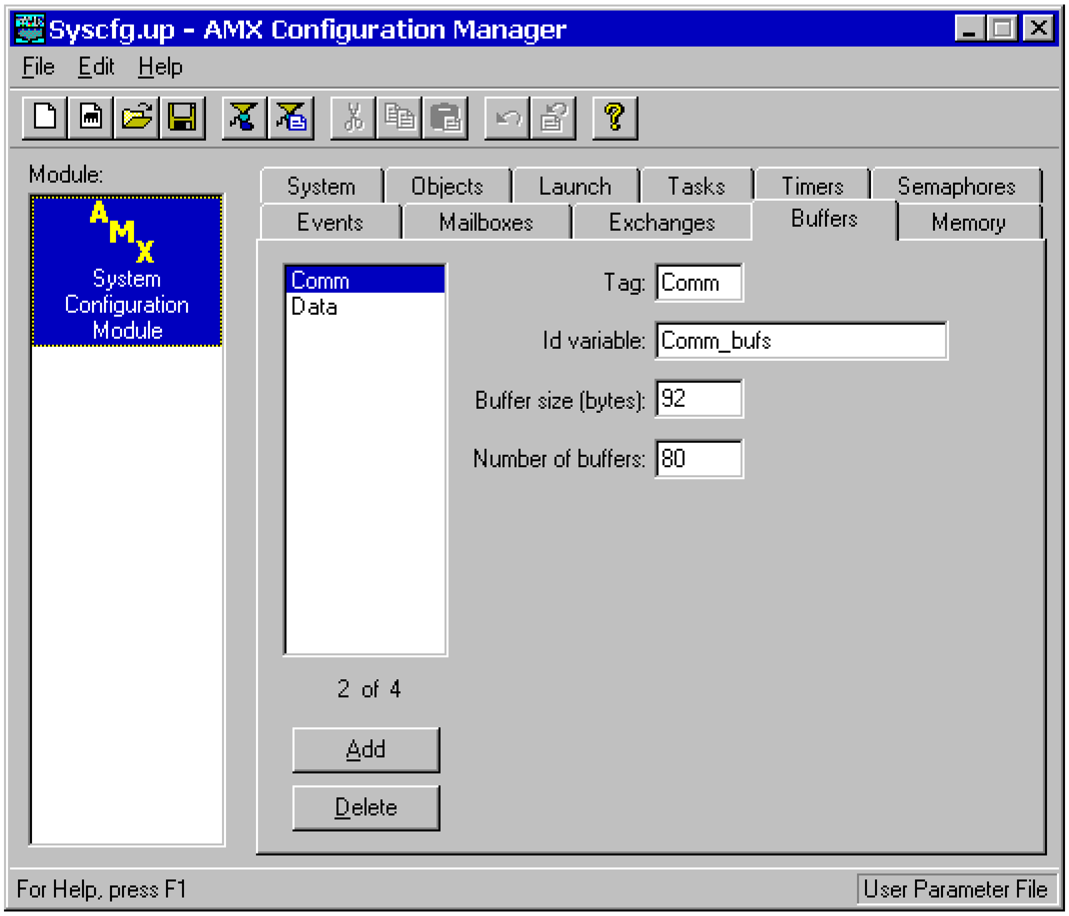
Message Queue Depths

These four parameters define the maximum number of message envelopes which can reside in each of the four message queues of the message exchange. Message queue 0 is the highest priority queue; queue 3 is the lowest priority.

Depths may range from 0 to 32767. If a particular message queue for a message exchange is to be unused, set its depth to zero. Other message queues may still be used. At least one message queue must be used. The maximum depth for a message queue does not affect AMX memory requirements.

Buffer Pool Definition

The Buffer Pool Definition window allows you to define the buffer pools to be automatically created by AMX at system startup. You do not have to predefine all of your buffer pools in this manner; you may also create buffer pools dynamically using *cjbmbuild* or *cjbmcreate*. The layout of the window is shown below.


Syscfg.up - AMX Configuration Manager
File Edit Help
Module:
AMX System Configuration Module
System Objects Launch Tasks Timers Semaphores
Events Mailboxes Exchanges Buffers Memory
Comm
Data
2 of 4
Add
Delete
For Help, press F1
User Parameter File

Tag: Comm
Id variable: Comm_bufs
Buffer size (bytes): 92
Number of buffers: 80

Tag

Each buffer pool can have a unique 4-character buffer pool tag. This parameter defines that tag. Although AMX does not restrict the content of the buffer pool tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *`CJ_ID`* in the System Configuration Module in which AMX will save the buffer pool id of the buffer pool.

Buffer Size

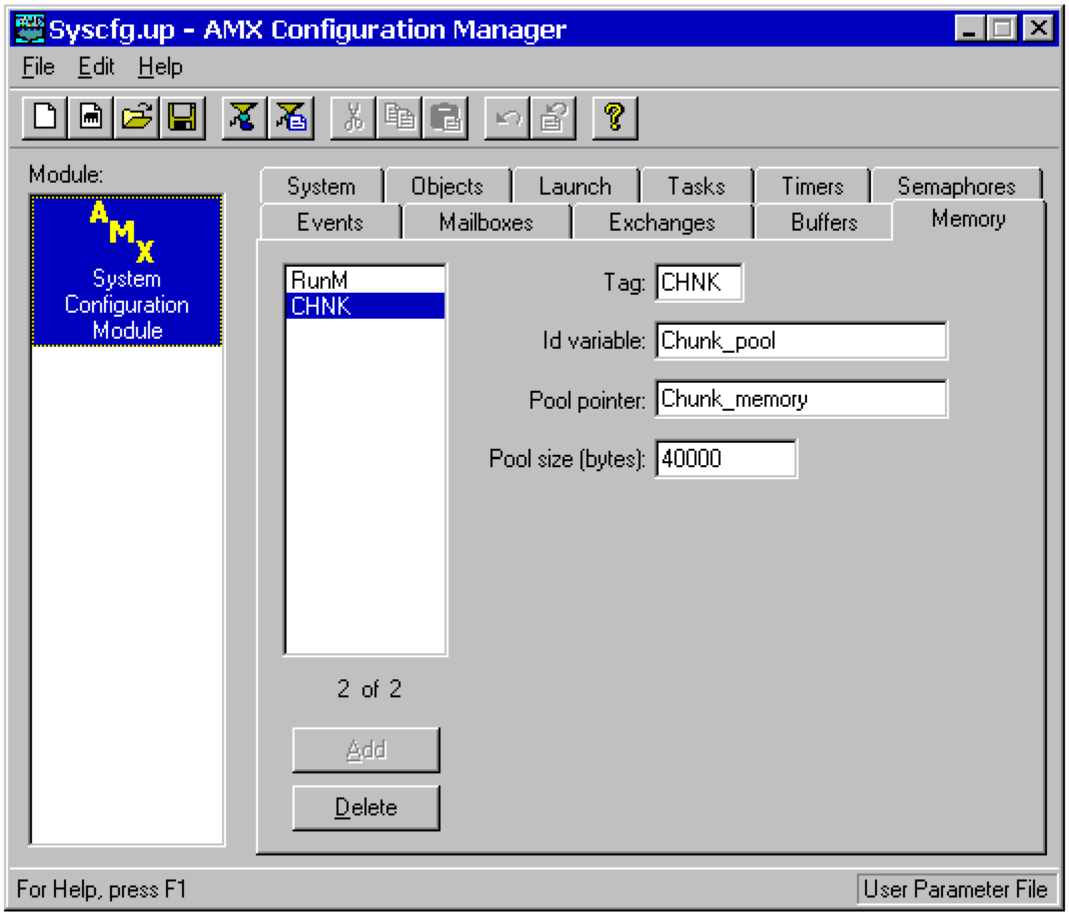
This parameter defines the useable size (in bytes) of each buffer in the buffer pool. If you enter a size of 0, the Manager will report the minimum allowed. The minimum buffer size is *`CJ_MINBFS`* bytes. The buffer size must be a multiple of 4.

Number of Buffers

This parameter defines the total number of buffers to be available in this buffer pool. The pool must contain at least one buffer.

Memory Pool Definition

The Memory Pool Definition window allows you to define the memory pools to be automatically created by AMX at system startup. You do not have to predefine all of your memory pools in this manner; you may also create memory pools dynamically using `cjmmbuild` or `cjmmcreate`. The layout of the window is shown below.



RunM
CHNK

2 of 2

Tag: CHNK

Id variable: Chunk_pool

Pool pointer: Chunk_memory

Pool size (bytes): 40000

Tag

Each memory pool can have a unique 4-character memory pool tag. This parameter defines that tag. Although AMX does not restrict the content of the memory pool tag field, the Configuration Manager only supports 4 ASCII characters as a tag.

Id Variable

This parameter defines the name of a public variable of type *CJ_ID* in the System Configuration Module in which AMX will save the memory pool id of the memory pool.

Memory Pool Pointer

This parameter defines the location of the memory to be assigned to the memory pool as a memory section. The location may be specified as the name of an external static character array variable of appropriate size and alignment (see *cjmmsection* description in Section 3).

Alternatively, an absolute hexadecimal address of the form *0xdddddddd* can be used to specify an explicit RAM address.

If this parameter is left blank, then the memory pool will be created by AMX but will have no memory for allocation until your application calls *cjmmsection* to assign one or more memory sections to the pool. If you leave this parameter blank, you must set the memory pool size to 0.

Memory Pool Size

This parameter defines the useable size (in bytes) of the memory section provided for the pool. If you enter a size of 1, the Manager will report the minimum allowed. The minimum memory pool size is *CJ_MINSMEM* bytes. The memory pool size must be a multiple of 4.

If the memory pool pointer is blank, the memory pool size must be set to 0.

This page left blank intentionally.

16. AMX Target Configuration

16.1 Target Configuration Module

The AMX Target Configuration Module defines the processor dependent hardware characteristics of your AMX system. The AMX Configuration Builder will create this module for you. This is the same tool used to create your AMX System Configuration Module (see Chapter 15.2).

The Target Configuration Module includes the following components.

The **Hardware Definition Table** provides AMX with all of the hardware specific information pertinent to the target processor which AMX requires for its proper operation. Most of the parameters in this table are predefined by KADAK. Some of the parameters may have to be defined by you to select the particular hardware features which you wish AMX to support in your AMX application. Such target dependent parameters are identified in each of the AMX Target Guides.

The Target Configuration Module includes the AMX **exception trap handlers** for all of the processor exceptions (interrupts) for which AMX is to be responsible.

The Target Configuration Module also includes the **clock ISP root** for the hardware clock interrupt to be used by AMX to derive its system tick. This ISP root can be one of the default clock ISPs provided with AMX or a custom clock ISP of your own making.

All **conforming ISPs** are located in the Target Configuration Module. The module must include an ISP root for each conforming device interrupt which your application intends to support. You must declare an ISP root for each device having an Interrupt Handler which expects to call AMX service procedures.

Note

The AMX Configuration Builder can be used to create your Target Configuration Module. Since this module varies significantly for each target processor, detailed instructions for using the Builder are provided in the target specific AMX Target Guide.

16.2 Target Configuration Generation

The AMX Configuration Builder is a software generation tool which can be used to create your AMX Target Configuration Module. The Builder helps to reduce total system implementation time by eliminating the manual generation process by which your Target Configuration Module would otherwise have to be produced. The Builder consists of two components: the Configuration Manager and the Configuration Generator. The Configuration Manager is an interactive utility which allows you to define and edit all of the parameters which must be included in your Target Configuration Module to use AMX on your partiular hardware platform.

The configuration process is illustrated in the block diagram of Figure 16.2-1.

The Configuration Manager lets you define your system to meet your needs. It produces a text file called a Target Parameter File. This file contains a cryptic representation of your description of your target hardware.

The Configuration Manager is also able to read a Target Parameter File allowing you to use the Configuration Manager to edit the content of a previously defined target description.

For convenience, the Configuration Manager has the ability to directly invoke its own copy of the Configuration Generator. The Configuration Generator reads your Target Parameter File and uses the information in it to produce a source file called the Target Configuration Module.

The Configuration Generator uses a file called the Target Configuration Template as a model for your Target Configuration Module. This template file is merged with the information in your Target Parameter File to produce your Target Configuration Module.

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.

If you are not using one of the toolsets supported by KADAK, you may still be able to use the AMX Configuration Builder by altering the Target Configuration Template File to meet the operating characteristics of your particular assembler.

The Configuration Generator is also available as a separate, stand alone DOS utility. This utility program can be used within your make files to generate and then assemble your Target Configuration Module. Instructions for doing so are provided in the AMX Tool Guide.

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.

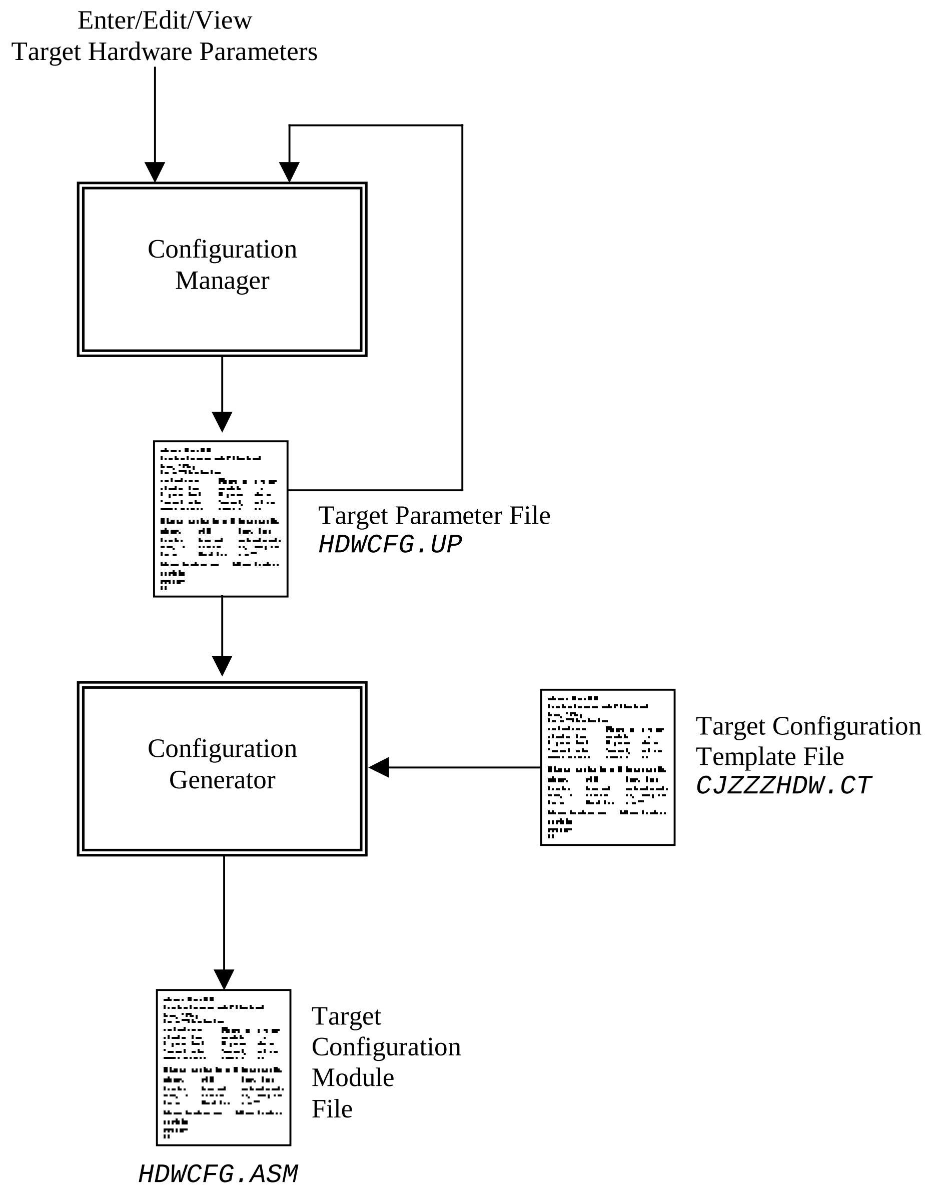


Figure 16.2-1 AMX Target Configuration Generation

Using the Builder

The AMX Configuration Builder will operate on a PC or compatible running the Microsoft® Windows® operating system.

The Builder is delivered with the following files.

File	Purpose
CJZZZCM .EXE	AMX Configuration Manager (utility program)
CJZZZCM .CNT	AMX Configuration Manager Help Content File
CJZZZCM .HLP	AMX Configuration Manager Help File
CJZZZCG .EXE	AMX Configuration Generator (utility program)
CJZZZHDW.CT	Target Configuration Template File

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, *CJZZZCM.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* into file *HDWCFG.UP* and 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.

The assembly language Target Configuration Module must be assembled as described in the toolset specific 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.

Note

Since the Target Configuration Module varies significantly for each target processor, detailed instructions for using the Builder are provided in the target specific AMX Target Guide.

16.3 Target Parameters

The AMX Configuration Builder can be used to create and edit your Target Parameter File. Since this file varies significantly for each target processor, a detailed description of the file is provided in the target specific AMX Target Guide. This chapter provides a brief introduction to the features which are common to most target processors.

The Target Parameter File is a text file structured as illustrated in Figure 16.3-1. The file consists of a sequence of keywords of the form `...XXX` which begin in column one. Each keyword is followed by one or more parameters recorded in the file by the Builder.

```
; Target hardware definitions
;
...HDW      PROC,<...>
;
:
;
; AMX launch parameters
:
...LAUNCH   PERM[,VNA]
;
:
;
; AMX Clock ISP
:
...CLKxxxx  <...>
:
;
;
; Your conforming ISP definitions
:
...ISPA     ISPROOT[,STEM],HANDLER,VNUM,PARAM,<...>
...ISPC     ISPROOT[,STEM],HANDLER,VNUM,PARAM,<...>
```

Figure 16.3-1 Target Parameter File

The example in Figure 16.3-1 uses symbolic names for all of the parameters following each of the keywords. The symbol names are replaced in your Target Parameter File with the actual parameters needed in your system. Parameters shown as `<...>` indicate that other target dependent parameters may be present. Parameters shown as `[,xxxx]` will only be present if required for a particular target processor.

The keywords listed in Figure 16.3-1 are present in the Target Parameter File for all target processors. Other target dependent keywords may exist and, if present, will be described in the target specific AMX Target Guide.

The order of keywords in the Target Parameter File is not particularly critical. For convenience, the keywords have been ordered to closely follow the order of the corresponding entries in your Target Configuration Module.

The Target Parameter File begins with a set of **hardware definitions**.

PROC	Processor identifier
<...>	Other target dependent parameters

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

PERM	AMX launch is temporary or permanent
[,VNA]	AMX Vector Table entries are or are not alterable

The **AMX Clock ISP** is identified using keyword *...CLKxxxx*. If you are using one of the clock ISPs provided with AMX for use with the target processor, the string *xxxx* will identify the particular clock type. The parameter list will provide parameters, if any, necessary to select the operating characteristics of the selected clock device.

If you choose to provide your own clock driver coded in C, the keyword *...CLKxxxx* will be replaced with *...CLKC*. If you choose to provide your own clock driver coded in assembly language, keyword *...CLKxxxx* will be replaced with *...CLKA*. In either case, the parameter list will define your conforming clock ISP root in exactly the same manner as for all other conforming ISPs.

You must declare a device ISP root for each **conforming ISP** which you intend to use in your application. For most AMX implementations, 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. For AMX PPC32, the ISP root is declared using *...ISPC* if its ISP stem is coded in C or *...ISPA* if its ISP stem is coded in assembly language. The parameter list is defined as follows.

ISPROOT	Name of the ISP root entry point
[,STEM]	Name of the public AMX PPC32 ISP stem
HANDLER	Name of the public device Interrupt Handler or of the public AMX PPC32 ISP Handler
VNUM	AMX vector number
PARAM	Name of a public variable whose address is to be passed as a parameter to the Interrupt Handler or to both the ISP stem and ISP Handler of an AMX PPC32 ISP
<...>	Other target dependent parameters

If *VNUM* is provided, AMX will automatically install the pointer to the ISP root *ISPROOT* into the appropriate entry in the Vector Table when AMX is launched. The vector initialization will be done before any application Restart Procedures are executed. If *VNUM* is *-1*, you must provide a Restart Procedure or task which installs the pointer to the ISP root *ISPROOT* into the Vector Table using one of the target specific AMX procedures *cjksivtwr* (or *cjksivtx*) or *cjksidtwr* (or *cjksidtx*).

Note that the AMX Configuration Builder unconditionally sets *VNUM* to *-1*, leaving it to your application to install the *ISPROOT* pointer into the AMX Vector Table.

17. AMX Service Procedures

17.1 Introduction

The AMX Library provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your final AMX system.

This section of the AMX User's Guide differs from the others because it is intended for use as a programming guide. The remainder of this chapter introduces you to the AMX programming environment. Chapter 18 provides descriptions of all of the procedures which are available in the AMX Runtime Library in alphabetic order for easy reference.

All of the AMX procedures are described using the C programming language. It is therefore recommended that you be at least superficially familiar with C.

A functional list of procedures is presented in Chapter 17.2. It is recommended that you use that chapter in conjunction with the procedure descriptions in Chapter 18 as follows. If you remember a procedure name but require information concerning its operation, go straight to the procedure description in the alphabetic list of procedures in Chapter 18. If you know what you wish to do functionally, but cannot remember the procedure name, go to Chapter 17.2 to quickly locate the procedure and then proceed to Chapter 18 to find its detailed description.

17.2 Summary of Services

AMX provides a wide variety of services from which the real-time system designer can choose. Many of the services are optional and, if not used, will not even be present in your AMX system. The AMX managers are all optional.

All of AMX and its managers are fully reentrant and may be placed in Read Only Memory (ROM).

The following lists summarize all of the AMX procedures which are accessible to the user. They are grouped functionally for easy reference.

Procedures are described in Chapter 18.

System Control Kernel Services (class *ks*)

cjkserror	AMX error or warning to User Error Procedure
cjksfatal	AMX Fatal Exit Procedure
cjksfind *cjksgbfind*	Find an AMX object with a specific tag
cjkshook	Install user hooks to AMX Scheduler
cjkslaunch *cjksenter*	Enter AMX multitasking world
cjksleave *cjksexit*	Leave AMX multitasking world
cjkspriv	Raise or lower task privilege level
cjksver	Get AMX version information

Task Control (class *tk*)

cjtkbuild	Create a new task (using a definition structure)
cjtkcreate	Create a new task (using inline parameters)
cjtkdelay	Delay for a timed interval
cjtkend	End current task execution
cjtkid	Get task id of current task
cjtkmsgack	Acknowledge receipt of a message
cjtkmxid	Get message exchange id for a message exchange task
cjtkmxinit	Initialize a message exchange task
cjtkpriority	Change a task's execution priority
cjtkpradjust	Sense and/or adjust a task's execution priority
cjtkresume	Resume a suspended task
cjtkstatus	Fetch task status
cjtksuspend	Suspend a task
cjtktcb	Get the Task Control Block pointer for a task
cjtktrigger	Trigger (start) a task
cjtkwait	Wait for a wake request
cjtkwaitclr	Reset any pending wake requests
cjtkwaitm	Timed wait for a wake request
cjtkwake	Wake a task

Task Termination Services (class *tk*)

cjtkdelete	Delete a task
cjtkkill	Kill a task Force a task to stop; flush all trigger requests
cjtkstop	Force a task to stop
cjtkterm	Enable/disable abnormal task termination
cjtkxdelete	Delete a message exchange task
cjtkxkill	Kill a message exchange task Force task to stop; flush all trigger requests Flush all message queues from task's private message exchange

Timing Control (class *tm*)

cjtmbuild	Create an interval timer (using a definition structure)
cjtmconvert	Convert milliseconds to system ticks
cjtmcreate	Create an interval timer (using inline parameters)
cjtmdelete	Delete an interval timer
cjtmread	Read an interval timer
cjtmslice	Change a task's time slice interval
cjtmtick	Read elapsed system ticks
cjtmtsopt	Enable/disable time slicing
cjtmwrite	Start/stop an interval timer

Time/Date Manager (class *td*)

cjtdfmt	Format calendar time/date as an ASCII string
cjtdget	Get calendar time/date
cjtdset	Set calendar time/date

Semaphore Manager (class *rm* or *sm*)

cjrmbuild	Create a resource semaphore (using a definition structure)
cjrmcreate	Create a basic resource semaphore (using inline parameters)
cjrmcreatex	Create an inheritance resource semaphore
cjrmdelete	Delete a resource semaphore
cjrmfree	Free a resource (unconditional)
cjrmrls	Release a resource (nested)
cjrmrsv	Reserve a resource (optional timeout)
cjrmstatus	Fetch resource semaphore status
cjsmbuild	Create a counting semaphore (using a definition structure)
cjsmcreate	Create a counting semaphore (using inline parameters)
cjsmdelete	Delete a counting semaphore
cjsmsignal	Signal to a counting semaphore
cjsmstatus	Fetch counting semaphore status
cjsmwait	Wait on a counting semaphore (optional timeout)

Event Manager (class *ev*)

cjevbuild	Create an event group (using a definition structure)
cjevcreate	Create an event group (using inline parameters)
cjevdelete	Delete an event group
cjevread	Read current state of events in an event group
cjevsignal	Signal one or more events in an event group
cjevstatus	Fetch event group status
cjevwait	Wait for all/any of a set of events in an event group (optional timeout)
cjevwaits	Get event flags at completion of event wait

Mailbox Manager (class *mb*)

cjmbbuild	Create a mailbox (using a definition structure)
cjmbcreate	Create a mailbox (using inline parameters)
cjmbdelete	Delete a mailbox
cjmbflush	Flush all messages or tasks from a mailbox
cjmbsend	Send a message to a mailbox (optional wait for acknowledgement)
cjmbstatus	Fetch mailbox status
cjmbwait	Wait for a message to arrive at a mailbox (optional timeout)

Message Exchange Manager (class *mx*)

cjmxbuild	Create a message exchange (using a definition structure)
cjmxcreate	Create a message exchange (using inline parameters)
cjmxdelete	Delete a message exchange
cjmxflush	Flush all messages or tasks from a message exchange
cjmxsend	Send a message to a message exchange (optional wait for acknowledgement)
cjmxstatus	Fetch message exchange status
cjmxwait	Wait for a message to arrive at a message exchange (optional timeout)

Buffer Manager (class *bm*)

cjbmbuild	Create a buffer pool (using a definition structure)
cjbmcreate	Create a buffer pool (using inline parameters)
cjbmdelete	Delete a buffer pool
cjbmfree	Free a buffer
cjbmget	Get a buffer from a specific pool (optional timeout)
cjbmid	Get a buffer's buffer pool id
cjbmsize	Get the size of a buffer
cjbmstatus	Fetch buffer pool status
cjbmuse	Add to a buffer's use count

Memory Manager (class *mm*)

cjmmbuild	Create a memory pool (using a definition structure)
cjmmcreate	Create a memory pool (using inline parameters)
cjmmdelete	Delete a memory pool
cjmmfree	Free a block of memory
cjmmget	Get a block of memory
cjmmid	Get a memory block's memory pool id
cjmmresize	Grow or shrink a memory block
cjmmsection	Add a section of memory to a memory pool
cjmmsize	Get the size of a block of memory
cjmmuse	Add to a memory block's use count

Circular List Manager (class *`cl`*)

`cjclabl`	Add to the bottom of a circular list
`cjclatl`	Add to the top of a circular list
`cjclinit`	Initialize a circular list
`cjclrbl`	Remove from the bottom of a circular list
`cjclrtl`	Remove from the top of a circular list

Linked List Manager (class *`lm`*)

`cjlmcreate`	Create an empty list
`cjlmhead`	Find the first object on a list (head)
`cjlminsc`	Insert before the specified object on a list
`cjlminsh`	Insert at the head of a list
`cjlminsk`	Insert into a keyed list
`cjlminst`	Insert at the tail of a list
`cjlmmerg`	Merge two lists
`cjlmnext`	Find the next object on a list (walk towards tail)
`cjlmordk`	Reorder an object in a keyed list
`cjlmprev`	Find the previous object on a list (walk towards head)
`cjlmrmvc`	Remove the specified object from a list
`cjlmrmvh`	Remove the object from the head of a list
`cjlmrmvt`	Remove the object from the tail of a list
`cjlmtail`	Find the last object on a list (tail)

Interrupt Control for AMX 68000, CFire, 4-ARM, 4-Thumb, PPC32, MA32 (class *ksi*)

cjksitrap	Install a task trap handler (if supported by processor)
cjksivtp	Fetch pointer to the Vector Table
cjksivtrd	Read an entry from the Vector Table
cjksivtwr	Write an entry into the Vector Table
cjksivtx	Exchange an entry in the Vector Table

Interrupt Control for AMX 386/ET (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. Refer to the appropriate AMX Target Guide for processor specific implementation details, including C functions for manipulating special processor registers.

cjcfccsetup	Setup C environment
cjcfdi	Disable interrupts
cjcfei	Enable interrupts
cjcfhwdelay	Delay *n* microseconds
cjcfhwXcache	Manipulate (flush/enable/disable) data and/or instruction caches
cjcfhwXflush	Flush data and/or instruction caches
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
cjcfstkjmp	Switch stacks and jump to a new procedure
cjcftag	Convert a string to an AMX tag value
cjcfvol8,16,32	Read a volatile 8, 16, 32-bit variable
cjcfvolpntr	Read a volatile pointer variable
	Hardware I/O port operations
cjcfinp8,16,32	Read from an 8, 16, 32-bit input I/O port
cjcfoutp8,16,32	Write an 8, 16, 32-bit value to an output I/O port
	Memory mapped I/O port operations
cjcfin8,16,32	Read from an 8, 16, 32-bit input I/O port
cjcfout8,16,32	Write an 8, 16, 32-bit value to an output I/O port

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 the AMX Target Guides and are subject to change at any time. The procedures are briefly described in source file *CJZZZUB.ASM* (or *CJZZZUB.S*). Prototypes will be found in file *CJZZZIF.H*. The register array structure *cjxregs* (or *cjxfpregs* or *cjxmregs*) which they reference is defined in file *CJZZZKT.H*.

cjcfregld	Load general registers from a register array
cjcfregst	Store general registers into a register array
cjcfsint	Generate a software initiated interrupt or exception
cjcffpregld	Load PowerPC floating point registers from a register array
cjcffpregst	Store PowerPC floating point registers into a register array
cjcfmregld	Load ARM banked registers from an extended register array
cjcfmregst	Store ARM banked registers into an extended register array

18. AMX Procedures

18.1 Introduction

A description of every AMX Library procedure is provided in this chapter. 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.

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

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

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

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

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

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

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

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

■ Disabled ■ Enabled (Not in ISP) ■ Restored

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

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

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

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

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

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

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

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

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

AMX PPC32 Users

AMX PPC32 does not actually disable interrupts to close the critical sections of code within its procedures. Instead, AMX PPC32 leaves the interrupt state unaltered but allows only quick interrupts during its critical operations. Refer to Chapter 3 of the AMX PPC32 Target Guide for a detailed descripton of the AMX interaction with the PowerPC™ interrupt system and a revised definition of the above interrupt legend.

Purpose **Build (Create) a Buffer Pool**

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmbuild(CJ_ID *bpidp, struct cjxbpdef *bpdefp);
```

Description *bpidp* is a pointer to storage for the buffer pool id of the buffer pool allocated to the caller.

bpdefp is a pointer to a buffer pool definition. Structure *cjxbpdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxbpdef {
  CJ_TAGDEF      xbpdtag;    /* Buffer pool tag           */
  void           *xbpdmemp;  /* Memory pointer            */
  long           xbpdmsize;  /* Size of memory            */
  int            xbpdnbuf;   /* Number of buffers         */
  int            xbpdbsize;  /* Size of buffers           */
  };
```

xbpdtag is a 4-character array for the buffer pool name tag.

xbpdmemp is a pointer to a long-aligned region of contiguous alterable memory (RAM) in which the Buffer Manager will create the buffer pool.

xbpdmsize is the size, in bytes, of the memory region referenced by *xbpdmemp*. The region *size* must be a multiple of 4 and must be at least *N = (xbpdbsize + sizeof(struct cjxbufh)) * xbpdnbuf*.

xbpdnbuf is the number of buffers in the buffer pool.

xbpdbsize is the usable size, in bytes, of each buffer in the pool. The buffer size must be a multiple of 4 and >= *CJ_MINBFS* (usually 8).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**bpidp* contains a valid buffer pool id.

Warnings returned:

CJ_WRBMMEMSIZ	Not enough memory provided

The memory region provided as input is not large enough to allocate *xbpdnbuf* buffers of size *xbpdbsize*. A valid buffer pool has been created and **bpidp* contains its buffer pool id. Use *cjbmstatus* to find the number of buffers actually allocated.

...more

Returns ...continued

Errors returned:

For all errors, the buffer pool id at `*bpidp` is undefined on return.

`CJ_ERBMNONE`	No free buffer pool
`CJ_ERBMNBUF`	No buffers defined in your pool definition
`CJ_ERBMSIZE`	Buffer size defined in your pool definition is too small

Example

```
#include "CJZZZ.H"
#define NBUF 10
#define BUFSIZE 64
#define POOLSIZE ((BUFSIZE + sizeof(struct cjxbufh)) * NBUF)

static long poolmemA[POOLSIZE/sizeof(long)];
static long poolmemB[POOLSIZE/sizeof(long)];

static struct cjxbpdef pooldefA = {
  {"Bp-A"},                /* Buffer pool tag           */
  poolmemA,                /* Memory pointer            */
  POOLSIZE,                /* Size of memory            */
  NBUF,                    /* Number of buffers         */
  BUFSIZE                  /* Size of buffers           */
  };


CJ_ID CJ_CCPP makepoolA(void) {
  CJ_ID poolid;

  if (cjbmbuild(&poolid, &pooldefA) >= CJ_EROK)
         return(poolid);    /* Accept warning            */
  else   return(CJ_IDNULL); /* Error                     */
  }


CJ_ID CJ_CCPP makepoolB(void) {
  struct cjxbpdef pooldefB;
  CJ_ID poolid;

  *(CJ_TYTAG *)&pooldefB.xbpdtag = cjcftag("Bp-B");
  pooldefB.xbpdmemp = poolmemB;
  pooldefB.xbpdmsize = POOLSIZE;
  pooldefB.xbpdnbuf = NBUF;
  pooldefB.xbpdbsize = BUFSIZE;

  if (cjbmbuild(&poolid, &pooldefB) >= CJ_EROK)
         return(poolid);    /* Accept warning            */
  else   return(CJ_IDNULL); /* Error                     */
  }
```

See Also `cjbmcreate, cjbmdelete, cjksfind`

Purpose **Create a Buffer Pool**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmcreate(CJ_ID *bpidp, char *tag,
                            void *memp, long memsize,
                            int nbuf, int bufsize);
```

Description *bpidp* is a pointer to storage for the buffer pool id of the buffer pool allocated to the caller.

tag is a pointer to a 4-character string for the buffer pool name tag.

memp is a pointer to a long-aligned region of contiguous alterable memory (RAM) in which the Buffer Manager will create the buffer pool.

memsize is the size, in bytes, of the memory region referenced by *memp*. The region length, in bytes, must be at least
*N = (bufsize + sizeof(struct cjxbufh)) * nbuf*.

nbuf is the number of buffers in the buffer pool.

bufsize is the usable size, in bytes, of each buffer in the pool. The buffer size must be a multiple of 4 and >= *CJ_MINBFS* (usually 8).

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**bpidp* contains a valid buffer pool id.

Warnings returned:

CJ_WRBMMEMSIZ	Not enough memory provided

The memory region provided as input is not large enough to allocate *nbuf* buffers of size *bufsize*. A valid buffer pool has been created and **bpidp* contains its buffer pool id. Use *cjbmstatus* to find the number of buffers actually allocated.

Errors returned:

For all errors, the buffer pool id at **bpidp* is undefined on return.

CJ_ERBMNONE	No free buffer pool
CJ_ERBMNBUF	No buffers defined in your pool definition
CJ_ERBMSIZE	Buffer size defined in your pool definition is too small

Example

```
#include "CJZZZ.H"
#define NBUF 10
#define BUFSIZE 64
#define POOLSIZE ((BUFSIZE + sizeof(struct cjxbufh)) * NBUF)

static long poolmemC[POOLSIZE/sizeof(long)];


CJ_ID CJ_CCPP makepoolC(void) {
   CJ_ID poolid;
   CJ_ERRST status;

   status = cjbmcreate(&poolid, "Bp-C",
            poolmemC, POOLSIZE, NBUF, BUFSIZE);

   if (status == CJ_EROK)
            return(poolid);

   else if (status == CJ_WRBMMEMSIZ)
            cjbmdelete(poolid);

   return(CJ_IDNULL);
   }
```

See Also *cjbmbuild, cjbmdelete, cjksfind*

Purpose	**Delete a Buffer Pool**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`CJ_ERRST CJ_CCPP cjbmdelete(CJ_ID poolid);`

Description	*`poolid`* is the buffer pool id of the buffer pool to be deleted.
Interrupts	■ Disabled ◻ Enabled ■ Restored
Returns	Error status is returned. *`CJ_EROK`* Call successful Errors returned: *`CJ_ERBMID`* Invalid buffer pool id *`CJ_ERBMBUSY`* Buffer pool is busy
Restrictions	You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the buffer pool. Failure to observe this restriction may lead to unexpected and unpredictable faults.
See Also	*`cjbmbuild, cjbmcreate`*

Purpose **Free a Buffer**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmfree(void *buffp);
```

Description *buffp* is a pointer to a buffer obtained by a *cjbmget* call.

Interrupts ■ Disabled ■ Enabled (Not in ISP) ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERBMBADP	Buffer pointer does not reference a valid buffer from a buffer pool
CJ_ERBMNOUSE	Buffer not in use
CJ_ERNOENVLOP	No message envelope available

The buffer use count is decremented by one. If the use count goes to zero, the buffer is returned to its buffer pool. If any tasks are waiting for a buffer from that pool, the buffer is given to the task at the head of the buffer pool's task wait list. Otherwise, the buffer is added to the end of the buffer pool's list of free buffers.

If the buffer use count does not go to zero, the buffer remains in use by other tasks in your system. The caller must not make any further references to the buffer.

Task Switch If the free buffer is given to a task waiting for a buffer, a task switch may occur. If the caller is a task, a task switch will occur if the waiting task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the waiting task is of higher priority than the interrupted task.

See Also *cjbmget, cjbmuse*

Purpose	**Get a Buffer**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmget(CJ_ID poolid, void *bufpp,
                         int priority, CJ_TIME timeout);
```

Description *poolid* is the id of the buffer pool from which the buffer is to be obtained.

bufpp is a pointer to storage for the returned pointer to the buffer. *bufpp* is prototyped as a *void ** allowing it to be a pointer to any type of pointer without the necessity of casts to keep some C compilers happy.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*. This parameter is not used if *timeout* is *< 0*.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for a buffer. If *timeout = 0*, the caller will wait forever for a buffer. If *timeout < 0*, the caller will not be allowed to wait for a buffer.

Interrupts ■ Disabled ■ Enabled (Not in ISP) ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**bufpp* contains a valid buffer pointer.
The buffer use count is set to one.

Warnings returned:
The content of **bufpp* is undefined.

CJ_WRBMNOBUF	No free buffer available
CJ_WRTMOUT	Timed out before buffer became available

Errors returned:
The content of **bufpp* is undefined.

CJ_ERBMID	Invalid buffer pool id
CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

Task Switch If the calling task waits for a buffer, there will be an immediate task switch to the next lower priority ready task.

Restrictions ISPs, Timer Procedures and Restart Procedures must call *cjbmget* with *timeout = -1* since they must not wait for a buffer.

See Also *cjbmfree, cjbmuse, cjbmsize, cjbmstatus*

Purpose **Get a Buffer's Buffer Pool Id**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmid(CJ_ID *bpidp, void *buffp);
```

Description *bpidp* is a pointer to storage for the buffer pool id of the buffer pool to which the buffer *buffp* belongs.

buffp is a pointer to a buffer obtained by a *cjbmget* call.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**bpidp* contains a valid buffer pool id.

Errors returned:

For all errors, the buffer pool id at **bpidp* is undefined on return.

CJ_ERBMBADP	Buffer pointer does not reference a valid buffer from a buffer pool

See Also *cjbmget, cjbmstatus*

Purpose	**Get Size of a Buffer**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmsize(void *buffp, int *sizep);
```

Description

buffp is a pointer to a buffer obtained by a *cjbmget* call.

sizep is a pointer to storage for the size, in bytes, of the buffer referenced by *buffp*.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns

Error status is returned.

CJ_EROK	Call successful

**sizep* contains the buffer size.

Errors returned:

For all errors, the buffer size at **sizep* is undefined on return.

CJ_ERBMBADP	Buffer pointer does not reference a valid buffer from a buffer pool
CJ_ERBMNOUSE	Buffer not in use

See Also *cjbmget, cjbmstatus*

Purpose **Get Status of a Buffer Pool**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmstatus(CJ_ID poolid,
                            struct cjxbpsts *statusp);
```

Description *poolid* is the buffer pool id of the buffer pool of interest.

statusp is a pointer to storage for the buffer pool status. Structure *cjxbpsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxbpsts {
   CJ_TYTAG      xbpstag;    /* Buffer pool tag           */
   int           xbpsnbuf;   /* Number of buffers in pool */
   int           xbpssize;   /* Buffer size (bytes)       */
   int           xbpsfree;   /* Number of free buffers    */
                             /* >0 # of buffers available */
                             /* 0  empty                  */
                             /* -n empty; n tasks waiting */
   };
```

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the buffer pool status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERBMID	Invalid buffer pool id

See Also *cjbmid, cjbmsize*

Purpose	**Add to a Buffer's Use Count**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjbmuse(void *buffp, int increment);
```

Description `buffp` is a pointer to a buffer obtained by a `cjbmget` call.

`increment` is the signed value to be added to the buffer use count.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Buffer use count = buffer use count + `increment`.

Errors returned:

For all errors, the buffer use count is left unaltered.

`CJ_ERBMBADP`	Buffer pointer does not reference a valid buffer from a buffer pool
`CJ_ERBMNOUSE`	Buffer not in use
`CJ_ERBMUSEOVF`	Buffer use count overflow or underflow The use count must remain > 0.

Once a buffer's use count is increased to *n*, the buffer will not be returned to the free list of its buffer pool until *n* calls to `cjbmfree` are made to release the buffer.

See Also `cjbmget, cjbmfree`

Purpose **Processor and C Interface Procedures**

Description The AMX Library includes several C procedures of a general nature which simplify application programming in real-time systems on your target processor. Refer to the appropriate AMX Target Guide for processor specific implementation details, including C functions for manipulating special processor registers.

cjcfccsetup	Setup C environment
cjcfdi	Disable interrupts
cjcfei	Enable interrupts
cjcfhwdelay	Delay *n* microseconds
cjcfhwXcache	Manipulate (flush/enable/disable) data and/or instruction caches
cjcfhwXflush	Flush data and/or instruction caches
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
cjcfstkjmp	Switch stacks and jump to a new procedure
cjcftag	Convert a string to an AMX tag value
cjcfvol8,16,32	Read a volatile 8, 16, 32-bit variable
cjcfvolpntr	Read a volatile pointer variable
	Hardware I/O port operations
cjcfinp8,16,32	Read from an 8, 16, 32-bit input I/O port
cjcfoutp8,16,32	Write an 8, 16, 32-bit value to an output I/O port
	Memory mapped I/O port operations
cjcfin8,16,32	Read from an 8, 16, 32-bit input I/O port
cjcfout8,16,32	Write an 8, 16, 32-bit value to an output I/O port

cjclabl
cjclatl

Purpose **Add to the Bottom or Top of a Circular List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
int CJ_CCPP cjclabl(struct cjxclist *clistp, CJ_T32U item);
int CJ_CCPP cjclatl(struct cjxclist *clistp, CJ_T32U item);
```

Description *clistp* is a pointer to the list header of a circular list previously initialized using *cjclinit*. The list header structure *cjxclist* is defined in file *CJZZZSD.H* as follows:

```
struct cjxclist {
  int          xclmax;     /* Maximum # items on list   */
                           /* merged with size of items */
  int          xclnum;     /* Current # items on list   */
  int          xcltop;     /* Index of current top      */
  int          xclbot;     /* Index of next bottom      */
  };
```

item is the 8, 16 or 32-bit item value to be added to the bottom or top of the list. The item size is determined by the circular list type which is established when the list is initialized by *cjclinit*.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Completion status is returned.

0	Added OK; list not full
1	Added OK; list now full
-1	Cannot add item; list is full

Example See example with *cjclinit*.

See Also *cjclinit, cjclrbl, cjclrtl*

Purpose **Initialize (Reset) a Circular List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjclinit(struct cjxclist *clistp,
                      int size, int nslot);
```

Description *clistp* is a pointer to storage to be used as a circular list. Since the size and structure of the circular list is known only to the caller, *clistp* points to the list header of the circular list to be initialized. The list header structure *cjxclist* is defined in file *CJZZZSD.H* as follows:

```
struct cjxclist {
   int          xclmax;    /* Maximum # items on list   */
                           /* merged with size of items */
   int          xclnum;    /* Current # items on list   */
   int          xcltop;    /* Index of current top      */
   int          xclbot;    /* Index of next bottom      */
   };
```

size is the slot size of the circular list (1, 2 or 4 corresponding to 8, 16 or 32-bit slots).

nslot is the number of slots in the list. *nslot* $< 2^{\text{intsize-2}}$ where intsize is the number of bits in an integer. For example, if an integer is 32 bits, *nslot* must be $< 2^{30}$.

Note A circular list is actually a private AMX structure consisting of two parts: a circular list header structure *cjxclist* followed by storage for *nslot* slots of *size* bytes each. The example illustrates two circular lists: a static list *clistA* and a dynamic list at **clistBp*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

...more

Example

```
#include "CJZZZ.H"
#define NSLOTA 32             /* Number of slots for list A*/
#define NSLOTB 512            /* Number of slots for list B*/

extern CJ_ID mempoolid;       /* Memory pool id            */

static struct cltypeA {       /* Allocate list A           */
  struct cjxclist clhead;
  char   clslots[NSLOTA];
  } clistA;

                              /* Pointer to dynamic list B */
static struct cjxclist *clistBp;

void CJ_CCPP clsample(void) {
  unsigned long memsize;      /* Actual size of list B     */
  char   ch;                  /* Characters from listA     */
  long   itemB;               /* Values from listB         */

                              /* Initialize list A         */
  cjclinit(&clistA.clhead,
         sizeof(clistA.clslots[0]), NSLOTA);

                              /* Get memory for list B     */
  cjmmget(mempoolid,
         sizeof(struct cjxclist) + (NSLOTB * sizeof(long)),
         &clistBp, &memsize);

                              /* Initialize list B         */
  cjclinit(clistBp, sizeof(long), NSLOTB);


  cjclabl(&clistA.clhead, 'A');
  cjclabl(&clistA.clhead, 'B');
  cjclabl(&clistA.clhead, 'C');

  cjclrtl(&clistA.clhead, &ch);               /* ch = 'A' */
  cjclrbl(&clistA.clhead, &ch);               /* ch = 'C' */
  cjclrbl(&clistA.clhead, &ch);               /* ch = 'B' */

  cjclatl(clistBp, 0x1L);
  cjclatl(clistBp, 0x2L);
  cjclatl(clistBp, 0x3L);

  cjclrbl(clistBp, &itemB);               /* itemB = 0x1L */
  cjclrtl(clistBp, &itemB);               /* itemB = 0x3L */
  cjclrbl(clistBp, &itemB);               /* itemB = 0x2L */
  }
```

See Also *cjclabl, cjclatl, cjclrbl, cjclrtl*

Purpose **Remove from the Bottom or Top of a Circular List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
int CJ_CCPP cjclrbl(struct cjxclist *clistp, void *itemp);
int CJ_CCPP cjclrtl(struct cjxclist *clistp, void *itemp);
```

Description *clistp* is a pointer to the list header of a circular list previously initialized using *cjclinit*. The list header structure *cjxclist* is defined in file *CJZZZSD.H* as follows:

```
struct cjxclist {
   int           xclmax;     /* Maximum # items on list   */
                             /* merged with size of items */
   int           xclnum;     /* Current # items on list   */
   int           xcltop;     /* Index of current top      */
   int           xclbot;     /* Index of next bottom      */
   };
```

itemp is a pointer to storage for the 8, 16 or 32-bit item value to be removed from the bottom or top of the list. The item size is determined by the circular list type which is established when the list is initialized by *cjclinit*.

If *itemp* is *NULL*, the item will be removed from the list but will not be returned to the caller.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Completion status is returned.

0	Removed OK; list not empty
1	Removed OK; list now empty
-1	Cannot remove item; list is empty

Example See example with *cjclinit*.

See Also *cjclinit, cjclabl, cjclatl*

Purpose **Build (Create) an Event Group**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevbuild(CJ_ID *evidp,
                           struct cjxevdef *evdefp);
```

Description *evidp* is a pointer to storage for the event group id of the event group allocated to the caller.

evdefp is a pointer to an event group definition. Structure *cjxevdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxevdef {
  CJ_TAGDEF     xevdtag;    /* Event group tag          */
  unsigned int xevdvalue;   /* Initial event values     */
  };
```

xevdtag is a 4-character array for the event group name tag.

xevdvalue is the initial value for all of the event flags in the event group.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK Call successful
**evidp* contains a valid event group id.

Errors returned:
For all errors, the event group id at **evidp* is undefined on return.
CJ_EREVNONE No free event group

...more

Example

```
#include "CJZZZ.H"

static struct cjxevdef groupdefA = {
   {"Ev-A"},                 /* Event group tag           */
   0x0005                    /* Initial event flags       */
   };


CJ_ID CJ_CCPP makegroupA(void) {
   CJ_ID groupid;

   if (cjevbuild(&groupid, &groupdefA) == CJ_EROK)
           return(groupid);
   else    return(CJ_IDNULL); /* Error                    */
   }


CJ_ID CJ_CCPP makegroupB(void) {
   struct cjxevdef groupdefB;
   CJ_ID groupid;

   *(CJ_TYTAG *)&groupdefB.xevdtag = cjcftag("Ev-B");
   groupdefB.xevdvalue = 0;

   if (cjevbuild(&groupid, &groupdefB) == CJ_EROK)
           return(groupid);
   else    return(CJ_IDNULL); /* Error                    */
   }
```

See Also *cjevcreate, cjevdelete, cjksfind*

Purpose **Create an Event Group**

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevcreate(CJ_ID *evidp, char *tag,
                            unsigned int value);
```

Description `evidp` is a pointer to storage for the event group id of the event group allocated to the caller.

`tag` is a pointer to a 4-character string for the event group name tag.

`value` is the initial value for all of the event flags in the event group.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

`*evidp` contains a valid event group id.

Errors returned:

For all errors, the event group id at `*evidp` is undefined on return.

`CJ_EREVNONE`	No free event group

Example

```
#include "CJZZZ.H"

CJ_ID CJ_CCPP makegroupC(void) {
  CJ_ID groupid;

  if (cjevcreate(&groupid, "Ev-C", 0x0005) == CJ_EROK)
         return(groupid);

  else   return(CJ_IDNULL);
  }
```

See Also `cjevbuild, cjevdelete, cjksfind`

Purpose	**Delete an Event Group**
Used by	■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`CJ_ERRST CJ_CCPP cjevdelete(CJ_ID groupid);`

Description	*`groupid`* is the event group id of the event group to be deleted.
Interrupts	■ Disabled □ Enabled ■ Restored
Returns	Error status is returned. *`CJ_EROK`* Call successful Errors returned: *`CJ_EREVID`* Invalid event group id *`CJ_EREVBUSY`* Event group is busy One or more tasks are still waiting for events in the group.
Restrictions	You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the event group. Failure to observe this restriction may lead to unexpected and unpredictable faults.
See Also	*`cjevbuild, cjevcreate`*

Purpose **Read the Current Event States in an Event Group**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevread(CJ_ID groupid, unsigned int *eventp);
```

Description *groupid* is the group id of the event group of interest.

eventp is a pointer to storage to receive the current state of the event flags of the particular event group.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**eventp* contains a copy of the event group flags.

Errors returned:

The content of **eventp* is undefined.

CJ_EREVID	Invalid event group id

Note *Cjevread* differs from *cjevwaits*. *Cjevread* gives you the current state of all event flags of a particular event group. *Cjevwaits* gives a task the state of all event flags as they were at the completion of that task's most recent call to *cjevwait*.

See Also *cjevstatus, cjevwaits*

Purpose **Signal Event(s) in an Event Group**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevsignal(CJ_ID groupid,
                            unsigned int mask,
                            unsigned int value, int type);
```

Description *groupid* is the group id of the event group of interest.

mask is a bit mask identifying the event flags of interest in the event group.

value is a bit pattern which specifies the desired states for each of the event flags selected by the mask. The values for flags not selected by the mask are ignored.

type defines the signal type: *CJ_EVCONST* for constant (level) events; *CJ_EVPULSE* for pulsed events. If *type = CJ_EVPULSE*, *value* must equal *mask*.

Constant events cause the selected flags to be adjusted to the values determined by *value*. Any task whose wait criteria is met is allowed to resume. The selected flags remain in the adjusted state.

Pulsed events cause a momentary transition of the selected flags to the one (true) state. Any task whose wait criteria is met is allowed to resume. The selected flags are then immediately reset.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_EREVID	Invalid event group id
CJ_ERNOENVLOP	Cannot signal the event(s) because no message envelopes are available for use

Task Switch If any tasks are waiting at the event group, an immediate task switch to the AMX Kernel Task will occur. If the caller is an ISP, the task switch will occur when the interrupt service is complete.

See Also *cjevwait*

Purpose **Get Status of an Event Group**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevstatus(CJ_ID groupid,
                            struct cjxevsts *statusp);
```

Description *groupid* is the event group id of the event group of interest.

statusp is a pointer to storage for the event group status. Structure *cjxevsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxevsts {
   CJ_TYTAG      xevstag;    /* Event group tag          */
   unsigned int xevsvalue;   /* Current event values     */
   int           xevscount;  /* Event group task count   */
                             /* >=0 if no tasks waiting  */
                             /* -n = n tasks waiting     */
   };
```

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the event group status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_EREVID	Invalid event group id

See Also *cjevwaits, cjevread*

Purpose **Wait for Event(s) in an Event Group**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjevwait(CJ_ID groupid,
                        unsigned int mask,
                        unsigned int value,
                        int match, CJ_TIME timeout);
```

Description *groupid* is the group id of the event group containing the events of interest.

mask is a bit mask identifying the event flags of interest in the group.

value is a bit pattern which specifies the states of interest for each of the event flags selected by the mask. The values for flags not selected by the mask are ignored. The value for pulsed event flags must be one.

match defines the event match requirements.

If *match = CJ_EVOR*, any selected flag in the state specified by *value* is considered the event of interest.

If *match = CJ_EVAND*, all selected flags must be in the state specified by *value* to be considered the event of interest.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for an event match. *Timeout* must be positive. If *timeout = 0*, the caller will wait forever for the specified event match. If *timeout < 0*, the caller will not be allowed to wait for an event match.

Interrupts ■ Disabled ■ Enabled ☐ Restored

Returns Error status is returned.

CJ_EROK	Call successful; event match occurred

Warnings returned:

CJ_WREVNOEVT	No event match; caller chose not to wait
CJ_WRTMOUT	Timed out before event match occurred

Errors returned:

CJ_EREVID	Invalid event group id
CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

...more

Returns ...continued

If the events in the group match your event selection criterion when you call `cjevwait`, the calling task continues execution immediately without waiting.

Upon return from `cjevwait` after a match or a timeout, the state of all event flags in the group at the time of the return are saved in the calling task's Task Control Block. Procedure `cjevwaits` can be called to retrieve the saved copy of the event flags. If you waited for any one of several events, you can interpret these flags to determine which of the events caused your task to resume. If you timed out waiting for all of several events to occur, you can interpret these flags to determine which events did not occur.

Task Switch If the calling task must wait for the events to occur, there will be an immediate task switch to the next lower priority ready task.

See Also `cjevstatus, cjevsignal, cjevwaits`

Purpose **Fetch a Task's Saved Event Flags**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
unsigned int CJ_CCPP cjevwaits(void);
```

Description Used by a task to fetch a copy of the event flags as they were at the completion of the task's most recent event wait call to *cjevwait*.

Interrupts □ Disabled □ Enabled □ Restored

Returns The calling task's saved event flag copy is returned.

A task can wait for events in an event group by calling *cjevwait*. *Cjevwaits* is used to retrieve the state of the event flags as they were at the time the *cjevwait* call completed.

Cjevwait saves a copy of the current state of all event flags in the group in the calling task's Task Control Block at the moment the calling task's event match occurs or its wait interval expires. Subsequent calls by the task to *cjevwaits* will retrieve these saved event flags.

Cjevwaits is paired with a task's most recent *cjevwait* call. *Cjevwait* only updates the saved event flag copy if an event match or timeout occurs. *Cjevwaits* will continue to return the same value until the task makes another call to *cjevwait* which results in an update of the saved event flag copy.

Note *Cjevwaits* differs from *cjevread*. *Cjevwaits* gives the calling task the state of all event flags as they were at the completion of the task's most recent call to *cjevwait*. *Cjevread* gives you the current state of all event flags of a particular event group.

See Also *cjevread, cjevsignal, cjevstatus, cjevwait*

Purpose **User Error Procedure**

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

This procedure is called by AMX whenever any AMX procedure detects an error condition.

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
int CJ_CCPP cjkserror(int error, CJ_ID taskid);
```

Description Procedure *cjkserror* is considered to be your application User Error Procedure. Source code for this procedure is provided in file *CJZZZUF.C*. Edit the procedure to meet your application's needs.

error is the AMX error code (*error < 0*) or warning code (*error > 0*) defining the error condition. Error codes are summarized in Appendix B.

taskid is the task id of the task which was executing at the time of the error. If an ISP was executing at the time of the error, *taskid* will be *CJ_IDNULL*.

Interrupts □ Disabled □ Enabled □ Restored

Returns The AMX error code *error*

The error code returned by *cjkserror* to AMX is subsequently returned by AMX to the caller of the AMX procedure in which the error condition was detected.

Note Your application can call *cjkserror*. Error codes <= *CJ_ERBASE* and warning codes >= *CJ_AKBASE* are reserved for use by your application.

See Also *cjksfatal*

Purpose **Fatal Exit Procedure**

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

This procedure is called by AMX whenever it detects a fatal error condition in which to proceed would invite disaster.

Setup Prototype is in file `CJZZZIF.H`.

```
#include "CJZZZ.H"
void CJ_CCPP cjksfatal(int error, CJ_ID taskid);
```

Description Procedure `cjksfatal` is considered to be your application Fatal Exit Procedure. Source code for this procedure is provided in file `CJZZZUF.C`. Edit the procedure to meet your application's needs.

`error` is the AMX fatal exit code defining the fatal condition. Fatal exit codes are summarized in Appendix B.

`taskid` is the task id of the task which was executing at the time of the fatal fault. If an ISP was executing at the time of the fatal fault, `taskid` will be `CJ_IDNULL`.

Interrupts ■ Disabled □ Enabled □ Restored

Returns There is no return from `cjksfatal`.

Note Your application can call `cjksfatal`. Fatal exit codes >= `CJ_FEBASE` are reserved for use by your application.

See Also `cjkserror`

Purpose **Find an AMX Object with a Specific Tag**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjksfind(CJ_ID *idp, char *tag,
                          CJ_IDKEY idkey);

CJ_ERRST CJ_CCPP cjksgbfind(CJ_ID *idp, CJ_TYTAG tagv,
                          CJ_IDKEY idkey);
```

Description *idp* is a pointer to storage for the object id of the general object of interest.

tag is a pointer to a 4-character name tag string identifying the object of interest.

tagv is a 32-bit name tag value of type *CJ_TYTAG* identifying the object of interest.

idkey is the object id key. Valid id keys are:

'B'	Buffer pool
'E'	Event group
'M'	Mailbox
'P'	Process (task)
'S'	counting Semaphore or resource Semaphore
'T'	Timer
'Y'	memorY pool
'X'	message eXchange

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK Call successful

**idp* contains the object id of an object of type *idkey* which matches the tag.

Errors returned:

For all errors, the object id at **idp* is undefined on return.

CJ_ERNOEXIST No object of type *idkey* with matching tag can be found.

If more than one object of type *idkey* was created with the same tag, you will get back the object id of one of them, but which one is not certain.

...more

Example

```
#include "CJZZZ.H"

/* Find a buffer pool given its name tag                  */

CJ_ID CJ_CCPP findpool(char *bptag) {
  CJ_ID poolid;

  if (cjksfind(&poolid, bptag, 'B') == CJ_EROK)
        return(poolid);

  else   return(CJ_IDNULL);
  }



/* Find task id for a task given its task definition      */

CJ_ID CJ_CCPP findtask(struct cjxtkdef *tkdefp) {
  CJ_ID taskid;

  if (cjksgbfind(&taskid,
                 *(CJ_TYTAG *)(&tkdefp->xtkdtag),
                 'P') == CJ_EROK)
        return(taskid);

  else   return(CJ_IDNULL);
  }
```

See Also All *cjXXbuild* and *cjXXcreate* procedures

Purpose **Install Task Scheduler Hooks**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjkshook(unsigned int hookmask,
                        void *hookp, struct cjxfhb *fhbp,
                        unsigned int key);
```

Description *hookmask* identifies the hooks which are to be installed. Set *hookmask* to *CJ_MAFNSHED* to install hooks into the AMX Task Scheduler.

hookp is a pointer to an array of four pointers to your scheduler procedures. The procedure pointers must be in the order illustrated in the example. Set *hookp* to *NULL* to remove the hooks.

fhbp is a pointer to a statically allocated function hook block for the private use of AMX. Structure *cjxfhb* is defined in header file *CJZZZSD.H*.

key is an ordering key used by AMX to determine the position of your hooks in the list of hooks which AMX maintains. Set *key* to *0xFFFF* to install application scheduler hooks.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Nothing

Example

```
#include "CJZZZ.H"
extern void starthook(void);
extern void endhook(void);
extern void suspendhook(void);
extern void resumehook(void);

static struct {
  void          (*uhstart)(void);
  void          (*uhend)(void);
  void          (*uhsuspend)(void);
  void          (*uhresume)(void);
  } userhooks = {starthook, endhook,
                 suspendhook, resumehook};

static struct cjxfhb userfhb;

void CJ_CCPP RRproc(void) {
  cjkshook(CJ_MAFNSHED, &userhooks, &userfhb, 0xFFFF);
  }
```

Restrictions Your hook procedures must be coded in assembler according to the rules established in the AMX Target Guide.

Purpose **Processor Dependent Interrupt Procedures**

Description The AMX Library includes a collection of C procedures for manipulating the content of the AMX Vector Table. These procedures are described in the target specific AMX Target Guide.

Interrupt Control for AMX 68000, AMX CFire, AMX 4-ARM, AMX 4-Thumb, AMX PPC32, AMX MA32

cjksitrap	Install a task trap handler (if supported by processor)
cjksivtp	Fetch pointer to the Vector Table
cjksivtrd	Read an entry from the Vector Table
cjksivtwr	Write an entry into the Vector Table
cjksivtx	Exchange an entry in the Vector Table

Interrupt Control for AMX 386/ET

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

Purpose **Launch (Enter) the AMX Multitasking System**

Used by This procedure must be called from your `main()` program to launch your AMX application.

Setup Prototype is in file `CJZZZIF.H`.

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

Prototype is in file `CJZZZKF.H`.

```
void CJ_CCPP cjksenter(CJ_CCONST1 struct cjxupt CJ_CCONST2 *uptp);
```

Description Procedure `cjkslaunch` is considered to be the startup code for your AMX application . Source code for this procedure is provided in file `CJZZZUF.C`. Edit the procedure to meet your application's startup needs.

After all application hardware startup initialization is complete, `cjkslaunch` must call `cjksenter` to start the AMX multitasking system.

AMX initializes its internal parameters, switches to its private stack, starts the AMX Kernel Task and executes all Restart Procedures. AMX then begins its multitasking operation.

`uptp` is a pointer to the User Parameter Table in your AMX System Configuration Module. Symbols `CJ_CCONSTx` are defined in header file `CJZZZCC.H` as `const` or blank to meet the varying syntax needs of different C compilers.

The structure `cjxupt` is defined in header file `CJZZZSD.H`.

Interrupts ■ Disabled ■ Enabled ■ Restored

Interrupts are disabled after you enter `cjksenter` while AMX initializes itself. AMX enables interrupts prior to calling each of your Restart Procedures. Interrupts remain enabled thereafter.

If your application shuts down and returns to `cjkslaunch`, interrupts are restored to their state at the time `cjkslaunch` was called.

Returns There is no return from `cjksenter` unless a task calls `cjksleave` to shutdown the AMX system. In that case, the error status `errcode` given to `cjksleave` is passed back to `cjkslaunch` for return to its caller.

If your AMX User Parameter Table does not include proper kernel data pointers, `cjkslaunch` returns fatal exit code `CJ_FECFG`.

See Also `cjksleave`

Purpose **Leave (Exit) the AMX Multitasking System**

Used by This procedure can only be called by a task which wishes to initiate an orderly shutdown of the AMX system.

Setup Prototype is in file *CJZZZIF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjksleave(int errcode, void *infop);
```

Prototype is in file *CJZZZKF.H*.

```
void CJ_CCPP cjksexit(void);
```

Description Procedure *cjksleave* is considered to be the shutdown code for your AMX application . Source code for this procedure is provided in file *CJZZZUF.C*. Edit the procedure to meet your application's shutdown needs.

errcode is an application dependent error status code which is passed back to procedure *cjkslaunch*.

infop is an additional application dependent pointer variable which is also passed back to *cjkslaunch*.

After passing the exit information back to *cjkslaunch*, procedure *cjksleave* must call *cjksexit* to force AMX to shut down your multitasking system. AMX executes all of your Exit Procedures allowing an orderly shutdown of the tasks in your system. AMX then returns, via *cjksenter*, to the launch procedure *cjkslaunch* which can complete your hardware shut down and restoral.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns There is no return from *cjksleave*. Execution resumes in procedure *cjkslaunch*.

Restrictions If AMX was configured for permanent execution and you call *cjksleave*, AMX will take its fatal exit via *cjksfatal* with fatal exit code *CJ_FENOEXIT*.

See Also *cjkslaunch, cjksfatal*

Purpose **Alter Task Privilege**

This procedure can be used to disable and restore AMX task switching by raising and lowering task privilege. A task's execution priority is not affected. Use of this procedure is discouraged since few applications will ever have the need to inhibit task switching.

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
void CJ_CCPP cjkspriv(int option);
```

Description `option = CJ_YES` to raise task privilege.

`option = CJ_NO` to lower task privilege.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Restrictions For every call to raise privilege there must be a subsequent call to lower privilege.

Once privilege has been raised, task switching is inhibited until privilege is lowered.

Calls to `cjkspriv` can be nested allowing privilege to be raised *n* times by nested procedures. There must be *n* subsequent calls to lower privilege. Task switching is only enabled again when the last of the calls to lower privilege is made.

Purpose **Get AMX Version Number**

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
unsigned long CJ_CCPP cjksver(void);
```

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns The AMX version number in the hexadecimal format `0xpp0vrrmmL` where:

pp	is the KADAK processor identification code	
v	is the major release number	(*1, 2, 3 ...*)
rr	is the major revision number	(*00, 01, 02 ...*)
mm	is the minor revision number	(*a, b, c ...*)

Purpose **Create an Empty List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlmcreate(struct cjxlh *listp, int offset);
```

Description *listp* is a pointer to the list header.

offset is the node offset (in bytes) at which the list node is located in application objects to be linked in this list.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Nothing

The list header is initialized to define an empty list.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier          */
  int    data;              /* Other application data   */
  struct cjxln listnode;    /* List node                */
  struct cjxlk keynode;     /* Key list node            */
  };

static struct cjxlh list;    /* List header              */
static struct cjxlh keylist; /* Key list header          */

#define NUMOBJ 12            /* Array of objects         */
static struct uobject objarray[NUMOBJ];


void CJ_CCPP listinit(void) {
  int    i;

                            /* Create empty lists       */
  cjlmcreate(&list,
         ((int) &( (struct uobject *)0 )->listnode) );
  cjlmcreate(&keylist,
         ((int) &( (struct uobject *)0 )->keynode) );

                            /* Fill lists with objects   */
  for (i = 0; i < NUMOBJ; i++) {
         cjlminst(&list, &objarray[i]);
         cjlminsk(&list, &objarray[i], NUMOBJ - i);
         }
  }
```

Purpose **Find First Object on List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmhead(struct cjxlh *listp);
```

Description *listp* is a pointer to the list header.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns A pointer to the object at the head of the list.

The object remains on the list.

Returns *NULL* if the list is empty.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier          */
  int    data;              /* Other application data   */
  struct cjxln listnode;    /* List node                */
  struct cjxlk keynode;     /* Key list node            */
  };

static struct cjxlh list;   /* List header              */


struct uobject * CJ_CCPP listhead(void) {

  return ( (struct uobject *) cjlmhead(&list) );
  }
```

Purpose **Insert Object before Current Object on List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlminsc(struct cjxlh *listp,
                        void *newobj, void *curobj);
```

Description *listp* is a pointer to the list header.

newobj is a pointer to the new object to be inserted before the object specified by *curobj*.

curobj is a pointer to a particular object on the list.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                 /* Item identifier          */
  int    data;               /* Other application data   */
  struct cjxln listnode;     /* List node                */
  struct cjxlk keynode;      /* Key list node            */
  };

static struct cjxlh list;    /* List header              */


void CJ_CCPP list2tail(struct uobject *objp) {
  struct uobject *tailp;     /* Tail pointer             */

                             /* Find object at tail      */
  tailp = (struct uobject *) cjlmtail(&list);

  if (tailp != NULL)
                             /* Insert object before tail */
        cjlminsc(&list, objp, tailp);
  }
```

Purpose **Insert Object at Head of List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlminsh(struct cjxlh *listp, void *objectp);
```

Description *listp* is a pointer to the list header.

objectp is a pointer to the object to be inserted as the new head of the list.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier            */
  int    data;              /* Other application data     */
  struct cjxln listnode;    /* List node                  */
  struct cjxlk keynode;     /* Key list node              */
  };

static struct cjxlh list;   /* List header                */


void CJ_CCPP listnewhead(struct uobject *objp) {

  cjlminsh(&list, objp);    /* Insert object at head      */
  }
```

Purpose **Insert Object into Keyed List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlminsk(struct cjxlh *listp,
                      void *objectp, unsigned int key);
```

Description *listp* is a pointer to the list header.

objectp is a pointer to the object to be inserted into the list.

key is the insertion key. Objects are inserted in order of ascending key values. The object with the smallest key will be at the head of the list. If other objects with the same key value already reside on the list, the new object will be inserted after all other objects with that key.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Restrictions The caller must own the list. The list must not be manipulated by other tasks, ISPs or Timer Procedures while this call is in progress. You can use the Semaphore Manager to control ownership of the list if necessary.

ISPs should avoid the use of this procedure unless dealing with very short lists. Use of this procedure with long lists may cause unacceptable timing effects in ISPs.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier          */
  int    data;              /* Other application data   */
  struct cjxln listnode;    /* List node                */
  struct cjxlk keynode;     /* Key list node            */
  };

static struct cjxlh keylist; /* Key list header         */


void CJ_CCPP listkeyadd(struct uobject *objp,
                   unsigned int key) {

  cjlminsk(&list, objp, key);
  }
```

See Also *cjlmordk*

Purpose **Insert Object at Tail of List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlminst(struct cjxlh *listp, void *objectp);
```

Description *listp* is a pointer to the list header.

objectp is a pointer to the object to be inserted as the new tail of the list.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier           */
  int    data;              /* Other application data    */
  struct cjxln listnode;    /* List node                 */
  struct cjxlk keynode;     /* Key list node             */
  };

static struct cjxlh list;   /* List header               */


void CJ_CCPP listnewtail(struct uobject *objp) {

  cjlminst(&list, objp);    /* Insert object at tail     */
  }
```

Purpose **Merge Two Lists**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlmmerg(struct cjxlh *destlist,
                        struct cjxlh *srclist,
                        void *destobj, void *srcobj);
```

Description *&destlist* is a pointer to the destination list header.

&srclist is a pointer to the source list header.

destobj is a pointer to the insertion point on the destination list.

srcobj is a pointer to the extraction point on the source list.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

All of the objects on the source list are inserted on the destination list before *destobj*. Objects are removed from the source list starting with *srcobj*, wrapping around from the tail to the head and ending with the object immediately previous to *srcobj*. This list of source objects is then inserted in order before *destobj*. The following example illustrates this process.

Before

```
              srcobj
srclist --- X -- Y -- Z
destlist -- A -- B -- C -- D -- E
                   destobj
```

After

```
srclist --- empty
destlist -- A -- B -- Y -- Z -- X -- C -- D -- E
```

Restrictions You must not merge two lists if either of the lists is a keyed list.

Purpose **Find Next Object on List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmnext(struct cjxlh *listp, void *curobj);
```

Description *listp* is a pointer to the list header.

curobj is a pointer to an object on this list.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns A pointer to the object on the list immediately following the object specified by *curobj*.

The objects remain on the list.

Returns *NULL* if *curobj* is at the tail of the list.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                 /* Item identifier           */
  int    data;               /* Other application data    */
  struct cjxln listnode;     /* List node                 */
  struct cjxlk keynode;      /* Key list node             */
  };

static struct cjxlh list;    /* List header               */


struct uobject * CJ_CCPP listscanfwd(int id) {

  struct uobject *objp;      /* Object pointer            */

  objp = cjlmhead(&list);    /* Find head of list         */

  while (objp != NULL) {
         if (objp->id == id)
                  break;     /* Found object of interest  */

         objp = (struct uobject *) cjlmnext(&list, objp);
         }

  return (objp);
  }
```

Purpose **Reorder an Object in a Keyed List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlmordk(struct cjxlh *listp,
                        void *objectp, unsigned int key);
```

Description *listp* is a pointer to the list header.

objectp is a pointer to the object on the list which is to be moved within the list.

key is the value of the new key for the object referenced by *objectp.*

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Restrictions The caller must own the list. The list must not be manipulated by other tasks, ISPs or Timer Procedures while this call is in progress. You can use the Semaphore Manager to control ownership of the list if necessary.

ISPs should avoid the use of this procedure unless dealing with very short lists. Use of this procedure with long lists may cause unacceptable timing effects in ISPs.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;              /* Item identifier          */
  int    data;            /* Other application data   */
  struct cjxln listnode;  /* List node                */
  struct cjxlk keynode;   /* Key list node            */
  };

static struct cjxlh keylist; /* Key list header       */


void CJ_CCPP listchgkey(struct uobject *objp,
                   unsigned int key) {

  cjlmordk(&keylist, objp, key);
  }
```

See Also *cjlminsk*

Purpose **Find Previous Object on List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmprev(struct cjxlh *listp, void *curobj);
```

Description *listp* is a pointer to the list header.

curobj is a pointer to an object on this list.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns A pointer to the object on the list immediately preceding the object specified by *curobj*.

The objects remain on the list.

Returns *NULL* if *curobj* is at the head of the list.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;               /* Item identifier          */
  int    data;             /* Other application data   */
  struct cjxln listnode;   /* List node                */
  struct cjxlk keynode;    /* Key list node            */
  };

static struct cjxlh list;  /* List header              */


struct uobject * CJ_CCPP listscanbwd(int id) {

  struct uobject *objp;    /* Object pointer           */

  objp = cjlmtail(&list);  /* Find tail of list        */

  while (objp != NULL) {
         if (objp->id == id)
                   break;  /* Found object of interest  */

         objp = (struct uobject *) cjlmprev(&list, objp);
         }

  return (objp);
  }
```

Purpose **Remove Object from List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void CJ_CCPP cjlmrmvc(struct cjxlh *listp, void *objectp);
```

Description *listp* is a pointer to the list header.

objectp is a pointer to the object to be removed from the list.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Restrictions The object referenced by *objectp* must be on the list referenced by *listp*. Failure to observe this requirement may lead to unpredictable side effects.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier          */
  int    data;              /* Other application data   */
  struct cjxln listnode;    /* List node                */
  struct cjxlk keynode;     /* Key list node            */
  };

static struct cjxlh list;   /* List header              */


struct uobject * CJ_CCPP listrmv(struct uobject *objp) {

  cjlmrmvc(&list, objp);    /* Remove object from list  */

  return (objp);
  }
```

Purpose **Remove Object from Head of List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmrmvh(struct cjxlh *listp);
```

Description *listp* is a pointer to the list header.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns A pointer to the object removed from the head of the list.

Returns *NULL* if the list was empty.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;               /* Item identifier           */
  int    data;             /* Other application data    */
  struct cjxln listnode;   /* List node                 */
  struct cjxlk keynode;    /* Key list node             */
  };

static struct cjxlh list;  /* List header               */


struct uobject * CJ_CCPP listrmvhead(void) {

                   /* Remove object from head of list    */
  return ( (struct uobject *) cjlmrmvh(&list) );
  }
```

Purpose **Remove Object from Tail of List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmrmvt(struct cjxlh *listp);
```

Description *listp* is a pointer to the list header.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns A pointer to the object removed from the tail of the list.

Returns *NULL* if the list was empty.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;                /* Item identifier          */
  int    data;              /* Other application data   */
  struct cjxln listnode;    /* List node                */
  struct cjxlk keynode;     /* Key list node            */
  };

static struct cjxlh list;   /* List header              */


struct uobject * CJ_CCPP listrmvtail(void) {

                   /* Remove object from tail of list     */
  return ( (struct uobject *) cjlmrmvt(&list) );
  }
```

Purpose **Find Last Object on List**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
void * CJ_CCPP cjlmtail(struct cjxlh *listp);
```

Description *listp* is a pointer to the list header.

Interrupts □ Disabled □ Enabled □ Restored

Returns A pointer to the object at the tail of the list.

The object remains on the list.

Returns *NULL* if the list is empty.

Example

```
#include "CJZZZ.H"

struct uobject {
  int    id;               /* Item identifier          */
  int    data;             /* Other application data   */
  struct cjxln listnode;   /* List node                */
  struct cjxlk keynode;    /* Key list node            */
  };

static struct cjxlh list;  /* List header              */


struct uobject * CJ_CCPP listtail(void) {

  return ( (struct uobject *) cjlmtail(&list) );
  }
```

Purpose **Build (Create) a Mailbox**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbbuild(CJ_ID *mbidp,
                           struct cjxmbdef *mbdefp);
```

Description *mbidp* is a pointer to storage for the mailbox id of the mailbox allocated to the caller.

mbdefp is a pointer to a mailbox definition. Structure *cjxmbdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxmbdef {
  CJ_TAGDEF     xmbdtag;    /* Mailbox tag              */
  int           xmbddepth;  /* Message queue depth      */
  };
```

xmbdtag is a 4-character array for the mailbox name tag.

xmbddepth is an integer defining the maximum number of message envelopes which can reside in the mailbox's message queue. (*0 <= xmbddepth <= 32767*).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mbidp* contains a valid mailbox id.

Errors returned:

For all errors, the mailbox id at **mbidp* is undefined on return.

CJ_ERMBNONE	No free mailbox
CJ_ERMBDEPTH	Invalid message queue depth

...more

Example

```
#include "CJZZZ.H"
#define DEPTH 32

static struct cjxmbdef mboxdefA = {
  {"Mb-A"},                 /* Mailbox tag              */
  DEPTH                     /* Message queue depth      */
  };


CJ_ID CJ_CCPP makeboxA(void) {
  CJ_ID mboxid;

  if (cjmbbuild(&mboxid, &mboxdefA) == CJ_EROK)
         return(mboxid);
  else   return(CJ_IDNULL); /* Error                    */
  }


CJ_ID CJ_CCPP makeboxB(void) {
  struct cjxmbdef mboxdefB;
  CJ_ID mboxid;

  *(CJ_TYTAG *)&mboxdefB.xmbdtag = cjcftag("Mb-B");
  mboxdefB.xmbddepth = DEPTH;

  if (cjmbbuild(&mboxid, &mboxdefB) == CJ_EROK)
         return(mboxid);
  else   return(CJ_IDNULL); /* Error                    */
  }
```

See Also *cjmbcreate, cjmbdelete, cjksfind*

Purpose	**Create a Mailbox**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbcreate(CJ_ID *mbidp, char *tag,
                            int depth);
```

Description

mbidp is a pointer to storage for the mailbox id of the mailbox allocated to the caller.

tag is a pointer to a 4-character string for the mailbox name tag.

depth is an integer defining the maximum number of message envelopes which can reside in the mailbox's message queue.
(*0 <= depth <= 32767*).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns

Error status is returned.

CJ_EROK	Call successful

**mbidp* contains a valid mailbox id.

Errors returned:

For all errors, the mailbox id at **mbidp* is undefined on return.

CJ_ERMBNONE	No free mailbox
CJ_ERMBDEPTH	Invalid message queue depth

Example

```
#include "CJZZZ.H"
#define DEPTH 32

CJ_ID CJ_CCPP makeboxC(void) {
  CJ_ID mboxid;

  if (cjmbcreate(&mboxid, "Mb-C", DEPTH) == CJ_EROK)
          return(mboxid);
  else    return(CJ_IDNULL); /* Error                    */
  }
```

See Also *cjmbbuild, cjmbdelete, cjksfind*

Purpose	**Delete a Mailbox**
Used by	■ Task □ ISP □ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbdelete(CJ_ID mboxid);
```

Description *mboxid* is the mailbox id of the mailbox to be deleted.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERMBID	Invalid mailbox id
CJ_ERMBBUSY	Mailbox is busy Messages may still be present in the mailbox or one or more tasks may be waiting for a message to arrive at the mailbox.

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the mailbox. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *cjmbbuild, cjmbcreate*

Purpose **Flush Messages From a Mailbox**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbflush(CJ_ID mboxid);
```

Description *mboxid* is the mailbox id of the mailbox to be flushed.

All message envelopes present in the mailbox message queue are released for reuse by AMX. The flushed messages are acknowledged with a warning status of *CJ_WRTKFLUSH* allowing the task which sent the message, if still waiting, to resume with the warning indication.

If the mailbox is empty but has one or more tasks waiting for a message, the tasks are forced to resume execution with the warning status *CJ_WRMBFLUSH*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERMBID	Invalid mailbox id

Restrictions The content of all flushed messages is lost. Do not flush any mailbox which, by design, could have messages present which contain such things as buffer pointers, memory block pointers or object ids which, if lost, could lead to lost resources within your application.

See Also *cjmbsend, cjmbwait*

Purpose	**Send a Message to a Mailbox**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbsend(CJ_ID mboxid, void *msgp, int wack);
```

Description

mboxid is the mailbox id of the mailbox of interest.

msgp is an array name or pointer to *CJ_MAXMSZ* consecutive bytes that form the message to be sent to the mailbox. The message variable must be integer aligned. *CJ_MAXMSZ* is configured in the User Parameter Table and is defined in file *CJZZZAPP.H*. For convenience, a maximum size AMX message structure *cjxmsg* is defined in file *CJZZZSD.H*.

wack is an integer which, if set to *CJ_YES*, indicates that the sending task wishes to wait for an acknowledgement that the message has been received by some other task. If *wack = CJ_NO*, then the sending task will not wait for acknowledgement. ISPs, Timer Procedures and Restart Procedures must always set *wack = CJ_NO*.

Interrupts ■ Disabled ■ Enabled (Not in ISP) ■ Restored

Returns

Error status is returned.

CJ_EROK	Call successful

If a task is waiting for the message, the *CJ_MAXMSZ* bytes of the message at **msgp* are copied directly to the waiting task's message buffer. Otherwise, the message is copied into a message envelope which is added to the end of the mailbox message queue.

Errors returned:

CJ_ERMBID	Invalid mailbox id
CJ_ERMBFULL	Mailbox is full
CJ_ERNOENVLOP	No free message envelope
CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

Message answer-back from task which received the message:

status > 0	Application defined value
CJ_WRTKFLUSH	Mailbox was flushed while task was waiting for message acknowledgement

...more

Task Switch If a task was waiting at the mailbox for a message, a task switch may occur. If the caller is a task, a task switch will occur if the waiting task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the waiting task is of higher priority than the interrupted task.

If the calling task waits for acknowledgement of its message, there will be an immediate task switch to the next lower priority ready task.

Example

```
#include "CJZZZ.H"
extern CJ_ID mboxA;          /* Mailbox A id              */

union umessage {
  struct cjxmsg umaxmsg;     /* Biggest AMX message       */
  struct usermsg umsg;       /* Define usermsg elsewhere  */
  };

CJ_ERRST CJ_CCPP sendmboxA(void) {
  CJ_ERRST        status;    /* Error status              */
  union umessage msg;        /* Message to send           */

                             /* Create message 1          */
  msg.umsg.msgnum = 1;
  msg.umsg.msgpntr = "Message number 1";
                             /* Send message 1            */
                             /* Do not wait for ack       */
  if ( (status = cjmbsend(mboxA, &msg, CJ_NO)) != CJ_EROK)
        return (status);

                             /* Create message 2          */
  msg.umsg.msgnum = 2;
  msg.umsg.msgpntr = "Message number 2";
                             /* Send message 2            */
                             /* Wait for ack              */
  status = cjmbsend(mboxA, &msg, CJ_YES);
  return (status);
  }
```

See Also *cjmbwait*

Purpose **Get Status of a Mailbox**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbstatus(CJ_ID mboxid,
                        struct cjxmbsts *statusp);
```

Description *mboxid* is the mailbox id of the mailbox of interest.

statusp is a pointer to storage for the mailbox status. Structure *cjxmbsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxmbsts {
  CJ_TYTAG      xmbstag;    /* Mailbox tag              */
  int           xmbsdepth;  /* Message queue depth      */
  int           xmbscount;  /* Message count            */
                            /* >0 # of messages in      */
                            /*    message queue         */
                            /* 0  empty                 */
                            /* -n empty; n tasks waiting */
  };
```

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the mailbox status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERMBID	Invalid mailbox id

Purpose **Wait for a Message from a Mailbox**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmbwait(CJ_ID mboxid, void *msgp,
                          int priority, CJ_TIME timeout);
```

Description *mboxid* is the mailbox id of the mailbox of interest.

msgp is an array name or pointer to *CJ_MAXMSZ* consecutive bytes of storage into which the message will be copied when available. The message storage variable must be integer aligned. *CJ_MAXMSZ* is configured in the User Parameter Table and is defined in file *CJZZZAPP.H*. For convenience, a maximum size AMX message structure *cjxmsg* is defined in file *CJZZZSD.H*.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for a message to arrive. If *timeout = 0*, the caller will wait forever for a message. If *timeout < 0*, the caller will not wait if a message is not immediately available.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**msgp* contains the *CJ_MAXMSZ* bytes of the message.

Warnings returned:

For all warnings, the message at **msgp* is undefined on return.

CJ_WRMBEMPTY	Mailbox is empty
CJ_WRTMOUT	Timed out before message available
CJ_WRMBFLUSH	Mailbox was flushed while task was waiting

Errors returned:

For all errors, the message at **msgp* is undefined on return.

CJ_ERMBID	Invalid mailbox id
CJ_ERAKNEED	Task has an unacknowledged answer-back message which it must acknowledge before it can get another message from any mailbox or message exchange.
CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

...more

Task Switch If the calling task waits for a message, there will be an immediate task switch to the next lower priority ready task.

Example

```
#include "CJZZZ.H"
extern CJ_ID mboxA;           /* Mailbox A id               */

union umessage {
   struct cjxmsg umaxmsg;     /* Biggest AMX message        */
   struct usermsg umsg;       /* Define usermsg elsewhere   */
   };


void CJ_CCPP waitmboxA(void) {
   union umessage msg;        /* Received message           */

                              /* Wait at priority 10        */
                              /* for up to 10 seconds       */
   if (cjmbwait(mboxA, &msg, 10, cjtmconvert(10000))
                  == CJ_EROK) {
          :
          :         /* Process message at msg.umsg          */
          :
                              /* Acknowledge message        */
          cjtkmsgack(CJ_AKBASE + 5);
          }
   }
```

See Also *cjmbsend, cjtkmsgack*

Purpose **Build (Create) a Memory Pool**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmbuild(CJ_ID *mpidp,
                           struct cjxmpdef *mpdefp);
```

Description *mpidp* is a pointer to storage for the memory pool id of the memory pool allocated to the caller.

mpdefp is a pointer to a memory pool definition. Structure *cjxmpdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxmpdef {
  CJ_TAGDEF     xmpdtag;    /* Memory pool tag          */
  void          *xmpdmemp;  /* Memory pointer           */
  unsigned long xmpdmsize;  /* Size of memory           */
  };
```

xmpdtag is a 4-character array for the memory pool name tag.

xmpdmemp is a pointer to a memory section to be assigned to the memory pool. The memory section must be a long-aligned region of contiguous alterable memory (RAM).

If the memory pool is to be created without any initial memory in the pool, set *xmpdmemp* to *NULL*. Memory sections must then be dynamically assigned to the memory pool using *cjmmsection*.

xmpdmsize is the size, in bytes, of the memory section referenced by *xmpdmemp*. The size must be a multiple of 4 and *>= CJ_MINSMEM* (usually 128). If no memory section is to be assigned to the memory pool (*xmpdmemp = NULL*), set *xmpdmsize = 0*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mpidp* contains a valid memory pool id.

Errors returned:

For all errors, the memory pool id at **mpidp* is undefined on return.

CJ_ERMMNONE	No free memory pool
CJ_ERMMALIGN	Memory section not long aligned
CJ_ERMMSIZE	Memory section size is too small

Example

```
#include "CJZZZ.H"
#define MEMSIZE (128*1024L)

static long CJ_CCHUGE poolmemA[MEMSIZE/sizeof(long)];
static long CJ_CCHUGE poolmemB[MEMSIZE/sizeof(long)];

static struct cjxmpdef pooldefA = {
  {"Mp-A"},                 /* Memory pool tag           */
  poolmemA,                 /* Memory pointer            */
  MEMSIZE                   /* Size of memory            */
  };


CJ_ID CJ_CCPP makepoolA(void) {
  CJ_ID poolid;

  if (cjmmbuild(&poolid, &pooldefA) == CJ_EROK)
         return(poolid);
  else   return(CJ_IDNULL); /* Error                     */
  }


CJ_ID CJ_CCPP makepoolB(void) {
  struct cjxmpdef pooldefB;
  CJ_ID poolid;

  *(CJ_TYTAG *)&pooldefB.xmpdtag = cjcftag("Mp-B");
  pooldefB.xmpdmemp = NULL;
  pooldefB.xmpdmsize = 0;

  if (cjmmbuild(&poolid, &pooldefB) == CJ_EROK) {
         cjmmsection(poolid, poolmemB, MEMSIZE);
         return(poolid);
         }
  else   return(CJ_IDNULL); /* Error                     */
  }
```

See Also *cjmmcreate, cjmmdelete, cjmmsection, cjksfind*

Purpose **Create a Memory Pool**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmcreate(CJ_ID *mpidp, char *tag,
                        void *memp, unsigned long memsize);
```

Description *mpidp* is a pointer to storage for the memory pool id of the memory pool allocated to the caller.

tag is a pointer to a 4-character string for the memory pool name tag.

memp is a pointer to a memory section to be assigned to the memory pool. The memory section must be a long-aligned region of contiguous alterable memory (RAM).

If the memory pool is to be created without any initial memory in the pool, set *memp* to *NULL*. Memory sections must then be dynamically assigned to the memory pool using *cjmmsection*.

memsize is the size, in bytes, of the memory section referenced by *memp*. The size must be a multiple of 4 and >= *CJ_MINSMEM* (usually 128). If no memory section is to be assigned to the memory pool (*memp = NULL*), set *memsize = 0.*

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mpidp* contains a valid memory pool id.

Errors returned:

For all errors, the memory pool id at **mpidp* is undefined on return.

CJ_ERMMNONE	No free memory pool
CJ_ERMMALIGN	Memory section not long aligned
CJ_ERMMSIZE	Memory section size is too small

...more

Example

```
#include "CJZZZ.H"
#define MEMSIZE (128*1024L)

static long CJ_CCHUGE poolmemC[MEMSIZE/sizeof(long)];


CJ_ID CJ_CCPP makepoolC(void) {
  CJ_ID poolid;

  if (cjmmcreate(&poolid, "Mp-C",
                  poolmemC, MEMSIZE) == CJ_EROK)
        return(poolid);
  else   return(CJ_IDNULL); /* Error                      */
  }
```

See Also *cjmmbuild, cjmmdelete, cjmmsection, cjksfind*

Purpose	**Delete a Memory Pool**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmdelete(CJ_ID poolid);
```

Description *`poolid`* is the memory pool id of the memory pool to be deleted.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERMMID`	Invalid memory pool id

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the memory pool. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *`cjmmbuild, cjmmcreate`*

Purpose **Free Previously Allocated Memory Block**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmfree(void *memp);
```

Description *memp* is a pointer to a memory block allocated by *cjmmget*.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERMMBADP	*memp* does not reference a valid memory block from a memory pool
CJ_ERMMNOUSE	Memory block not in use

The memory block use count is decremented by one. If the use count goes to zero, the memory block is returned to its memory pool. The memory block is immediately coalesced with adjacent free memory blocks.

If the memory block use count does not go to zero, the memory block remains in use by other tasks in your system. The caller must not make any further references to the memory block.

See Also *cjmmget, cjmmuse*

Purpose	**Get a Memory Block**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmget(CJ_ID poolid, unsigned long size,
                         void *mempp, unsigned long *sizep);
```

Description *poolid* is the id of the memory pool from which the memory block is to be obtained.

size is the number of bytes of memory required.

mempp is a pointer to storage for the returned pointer to the memory block. *mempp* is prototyped as a *void ** allowing it to be a pointer to any type of pointer without the necessity of casts to keep some C compilers happy.

sizep is a pointer to storage for the actual usable size in bytes of the allocated memory block. **sizep* may be slightly larger than *size*. It is valid to replace *sizep* with *&size* to update your *size* variable.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mempp* contains a valid memory block pointer.
The memory block use count is set to one.

Warnings returned:
The content of **bufpp* is undefined.

CJ_WRMMNOMEM	No free memory block available **sizep* is the size of the largest currently available block of memory.

Errors returned:
The content of **mempp* and **sizep* is undefined.

CJ_ERMMID	Invalid memory pool id

See Also *cjmmfree, cjmmuse, cjmmresize, cjmmsize*

Purpose	**Get a Memory Block's Memory Pool Id**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmid(CJ_ID *mpidp, void *memp);
```

Description *`mpidp`* is a pointer to storage for the memory pool id of the memory pool to which the memory block *`memp`* belongs.

`memp` is a pointer to a memory block obtained by a *`cjmmget`* call.

Interrupts □ Disabled □ Enabled □ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

`*mpidp` contains a valid memory pool id.

Errors returned:

For all errors, the memory pool id at *`*mpidp`* is undefined on return.

`CJ_ERMMBADP`	*`memp`* does not reference a valid memory block from a memory pool

See Also *`cjmmget`*

Purpose **Change Size of a Memory Block**

You can grow or shrink a memory block. A memory block can always be shrunk. A memory block can only be increased in size if there is free memory immediately beyond the current end of the memory block.

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmresize(void *memp, unsigned long size);
```

Description *memp* is a pointer to a memory block allocated by *cjmmget*.

size is the required final size for the memory block. If *size* is smaller than the minimum allowable block size, the memory block will shrink to the minimum size allowed.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERMMBADP	*memp* does not reference a valid memory block from a memory pool
CJ_ERMMGROW	Cannot grow memory block The adjacent memory is not free or, if free, is too small.

See Also *cjmmget, cjmmsize*

Purpose **Add a Memory Section to a Memory Pool**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmsection(CJ_ID poolid, void *memp,
                             unsigned long memsize);
```

Description *poolid* is the id of the memory pool to which the memory section is to be assigned.

memp is a pointer to a memory section to be assigned to the memory pool. The memory section must be a long-aligned region of contiguous alterable memory (RAM).

memsize is the size, in bytes, of the memory section referenced by *memp*. The size must be a multiple of 4 and `>= CJ_MINSMEM` (usually 128).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERMMID	Invalid memory pool id
CJ_ERMMALIGN	Memory section not long aligned
CJ_ERMMSIZE	Memory section size is too small

See Also *cjmmbuild, cjmmcreate*

Purpose	**Get Size of a Memory Block**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`CJ_ERRST CJ_CCPP cjmmsize(void *memp, unsigned long *sizep);`

Description	*`memp`* is a pointer to a memory block allocated by `cjmmget`. *`sizep`* is a pointer to storage for the size, in bytes, of the memory block referenced by *`memp`*.
Interrupts	☐ Disabled ☐ Enabled ☐ Restored
Returns	Error status is returned. *`CJ_EROK`* Call successful *`*sizep`* contains the memory block size. Errors returned: For all errors, the memory block size at *`*sizep`* is undefined on return. *`CJ_ERMMBADP`* *`memp`* does not reference a valid memory block from a memory pool *`CJ_ERMMNOUSE`* Memory block not in use
See Also	*`cjmmget, cjmmresize`*

Purpose **Add to a Memory Block's Use Count**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmmuse(void *memp, int increment);
```

Description *memp* is a pointer to a memory block obtained by a *cjmmget* call.

increment is the signed value to be added to the memory block use count.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Memory block use count = memory block use count + *increment*.

Errors returned:

For all errors, the memory block use count is left unaltered.

CJ_ERMMBADP	*memp* does not reference a valid memory block from a memory pool
CJ_ERMMNOUSE	Memory block not in use
CJ_ERMMUSEOVF	Memory block use count overflow or underflow The use count must remain > 0.

Once a memory block's use count is increased to *n*, the memory block will not be returned to the free list of its memory pool until *n* calls to *cjmmfree* are made to release the memory block.

See Also *cjmmget, cjmmfree*

Purpose **Build (Create) a Message Exchange**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxbuild(CJ_ID *mxidp,
                           struct cjxmxdef *mxdefp);
```

Description *mxidp* is a pointer to storage for the message exchange id of the message exchange allocated to the caller.

mxdefp is a pointer to a message exchange definition. Structure *cjxmxdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxmxdef {
  CJ_TAGDEF     xmxdtag;    /* Message exchange tag      */
                            /* Message queue depths      */
  int xmxddepth[CJ_MAXMXBOX];
  };
```

xmxdtag is a 4-character array for the message exchange name tag.

xmxddepth is an array of *CJ_MAXMXBOX* (4) integers defining the maximum number of message envelopes which can reside in each of the message exchange's message queues.
(*0 <= xmxddepth <= 32767*).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mxidp* contains a valid message exchange id.

Errors returned:
For all errors, the message exchange id at **mxidp* is undefined on return.

CJ_ERMXNONE	No free message exchange
CJ_ERMXDEPTH	Invalid message queue depth

...more

Example

```
#include "CJZZZ.H"
#define DEPTH0 32
#define DEPTH3 16

static struct cjxmxdef msgxdefA = {
  {"Mx-A"},                 /* Message exchange tag       */
  DEPTH0, 0, 0, DEPTH3      /* Message queue depths       */
  };


CJ_ID CJ_CCPP makemsgxA(void) {
  CJ_ID msgxid;

  if (cjmxbuild(&msgxid, &msgxdefA) == CJ_EROK)
        return(msgxid);
  else   return(CJ_IDNULL); /* Error                      */
  }


CJ_ID CJ_CCPP makemsgxB(void) {
  struct cjxmxdef msgxdefB;
  CJ_ID msgxid;

  *(CJ_TYTAG *)&msgxdefB.xmxdtag = cjcftag("Mx-B");
  msgxdefB.xmxddepth[0] = DEPTH0;
  msgxdefB.xmxddepth[1] = 0;
  msgxdefB.xmxddepth[2] = 0;
  msgxdefB.xmxddepth[3] = DEPTH3;

  if (cjmxbuild(&msgxid, &msgxdefB) == CJ_EROK)
        return(msgxid);
  else   return(CJ_IDNULL); /* Error                      */
  }
```

See Also *cjmxcreate, cjmxdelete, cjksfind*

Purpose **Create a Message Exchange**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxcreate(CJ_ID *mxidp, char *tag,
                        int depth0, int depth1,
                        int depth2, int depth3);
```

Description *mxidp* is a pointer to storage for the message exchange id of the message exchange allocated to the caller.

tag is a pointer to a 4-character string for the message exchange name tag.

depthn are integers defining the maximum number of message envelopes which can reside on each of the message exchange's message queues. (*0 <= depthn <= 32767*).

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**mxidp* contains a valid message exchange id.

Errors returned:

For all errors, the message exchange id at **mxidp* is undefined on return.

CJ_ERMXNONE	No free message exchange
CJ_ERMXDEPTH	Invalid message queue depth

Example

```
#include "CJZZZ.H"
#define DEPTH0 32
#define DEPTH3 16

CJ_ID CJ_CCPP makemsgxC(void) {
  CJ_ID msgxid;

  if (cjmxcreate(&msgxid, "Mx-C", DEPTH0, 0, 0, DEPTH3)
                 == CJ_EROK)
         return(msgxid);
  else   return(CJ_IDNULL); /* Error                     */
  }
```

See Also *cjmxbuild, cjmxdelete, cjksfind*

Purpose	**Delete a Message Exchange**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`CJ_ERRST CJ_CCPP cjmxdelete(CJ_ID msgxid);`

Description	*`msgxid`* is the message exchange id of the message exchange to be deleted.
Interrupts	■ Disabled ◻ Enabled ■ Restored
Returns	Error status is returned. *`CJ_EROK`* Call successful Errors returned: *`CJ_ERMXID`* Invalid message exchange id *`CJ_ERMXBUSY`* Message exchange is busy Messages may still be present in the message exchange or one or more tasks may be waiting for a message to arrive at the message exchange.
Restrictions	You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.
See Also	*`cjmxbuild, cjmxcreate`*

Purpose	**Flush Messages From a Message Exchange**
Used by	■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxflush(CJ_ID msgxid);
```

Description *`msgxid`* is the message exchange id of the message exchange to be flushed.

All message envelopes present in any of the message exchange's message queues are released for reuse by AMX. The flushed messages are acknowledged with a warning status of *`CJ_WRTKFLUSH`* allowing the task which sent the message, if still waiting, to resume with the warning indication.

If the message exchange is empty but has one or more tasks waiting for a message, the tasks are forced to resume execution with the warning status *`CJ_WRMXFLUSH`*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERMXID`	Invalid message exchange id

Restrictions The content of all flushed messages is lost. Do not flush any message exchange which, by design, could have messages present which contain such things as buffer pointers, memory block pointers or object ids which, if lost, could lead to lost resources within your application.

See Also *`cjmxsend, cjmxwait`*

Purpose	**Send a Message to a Message Exchange**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxsend(CJ_ID msgxid, void *msgp,
                          int wack, int msgpriority);
```

Description *msgxid* is the message exchange id of the message exchange of interest.

msgpriority is the priority of the message.
(0 = highest; 3 = lowest).

msgp is an array name or pointer to *CJ_MAXMSZ* consecutive bytes that form the message to be sent to the message exchange. The message variable must be integer aligned. *CJ_MAXMSZ* is configured in the User Parameter Table and is defined in file *CJZZZAPP.H*. For convenience, a maximum size AMX message structure *cjxmsg* is defined in file *CJZZZSD.H*.

wack is an integer which, if set to *CJ_YES*, indicates that the sending task wishes to wait for an acknowledgement that the message has been received by some other task. If *wack = CJ_NO*, then the sending task will not wait for acknowledgement. ISPs, Timer Procedures and Restart Procedures must always set *wack = CJ_NO*.

Interrupts ■ Disabled ■ Enabled (Not in ISP) ■ Restored

Returns Error status is returned.

CJ_EROK Call successful

If a task is waiting for the message, the *CJ_MAXMSZ* bytes of the message at **msgp* are copied directly to the waiting task's message buffer. Otherwise, the message is copied into a message envelope which is added to the end of the message exchange message queue of priority *msgpriority*.

Errors returned:

CJ_ERMXID	Invalid message exchange id
CJ_ERMXMBNUM	Invalid message priority (must be 0 to 3)
CJ_ERMXFULL	The message queue at priority *msgpriority* is full
CJ_ERNOENVLOP	No free message envelope
CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

...more

Returns ...continued

Message answer-back from task which received the message:

`status > 0`	Application defined value
`CJ_WRTKFLUSH`	Message exchange was flushed while task was waiting for message acknowledgement

Task Switch If a task was waiting at the message exchange for a message, a task switch may occur. If the caller is a task, a task switch will occur if the waiting task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the waiting task is of higher priority than the interrupted task.

If the calling task waits for acknowledgement of its message, there will be an immediate task switch to the next lower priority ready task.

Example

```
#include "CJZZZ.H"
extern CJ_ID msgxA;            /* Message exchange A id     */

union umessage {
   struct cjxmsg umaxmsg;      /* Biggest AMX message       */
   struct usermsg umsg;        /* Define usermsg elsewhere  */
   };


CJ_ERRST CJ_CCPP sendmsgxA(void) {
   CJ_ERRST         status;    /* Error status              */
   union umessage msg;         /* Message to send           */

                               /* Create message 1          */
   msg.umsg.msgnum = 1;
   msg.umsg.msgpntr = "Message number 1";
                               /* Send message 1; priority 0*/
                               /* Do not wait for ack       */
   if ( (status = cjmxsend(msgxA, &msg, CJ_NO, 0))
                    != CJ_EROK)
         return (status);

                               /* Create message 2          */
   msg.umsg.msgnum = 2;
   msg.umsg.msgpntr = "Message number 2";
                               /* Send message 2; priority 3*/
                               /* Wait for ack              */
   status = cjmxsend(msgxA, &msg, CJ_YES, 3);
   return (status);
   }
```

See Also *cjmxwait*

Purpose **Get Status of a Message Exchange**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxstatus(CJ_ID msgxid,
                        struct cjxmxsts *statusp);
```

Description *msgxid* is the message exchange id of the message exchange of interest.

statusp is a pointer to storage for the message exchange status. Structure *cjxmxsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxmxsts {
  CJ_TYTAG      xmxstag;    /* Message exchange tag      */
  int           xmxsqueue;  /* Message availability      */
                            /* >0 # of messages in       */
                            /*    message queues         */
                            /* 0  all queues empty       */
                            /* -n empty; n tasks waiting */
                            /* Message queue depths      */
  int xmxsdepth[CJ_MAXMXBOX];
                            /* Message queue counts      */
  int xmxscount[CJ_MAXMXBOX];
  };
```

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the message exchange status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERMXID	Invalid message exchange id

Purpose **Wait for a Message from a Message Exchange**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjmxwait(CJ_ID msgxid, void *msgp,
                          int priority, CJ_TIME timeout);
```

Description *msgxid* is the message exchange id of the message exchange of interest.

msgp is an array name or pointer to *CJ_MAXMSZ* consecutive bytes of storage into which the message will be copied when available. The message storage variable must be integer aligned. *CJ_MAXMSZ* is configured in the User Parameter Table and is defined in file *CJZZZAPP.H*. For convenience, a maximum size AMX message structure *cjxmsg* is defined in file *CJZZZSD.H*.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for a message to arrive. If *timeout = 0*, the caller will wait forever for a message. If *timeout < 0*, the caller will not wait if a message is not immediately available.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**msgp* contains the *CJ_MAXMSZ* bytes of the message.

Warnings returned:

For all warnings, the message at **msgp* is undefined on return.

CJ_WRMXEMPTY	Message exchange is empty
CJ_WRTMOUT	Timed out before message available
CJ_WRMXFLUSH	Message exchange was flushed while task was waiting

...more

Returns ...continued

Errors returned:

For all errors, the message at *`*msgp`* is undefined on return.

`CJ_ERMXID`	Invalid message exchange id
`CJ_ERAKNEED`	Task has an unacknowledged answer-back message which it must acknowledge before it can get another message from any mailbox or message exchange.
`CJ_ERSMUV`	Resource semaphore usage violation (see *`cjtkwait`*)

Task Switch If the calling task waits for a message, there will be an immediate task switch to the next lower priority ready task.

Example

```
#include "CJZZZ.H"
extern CJ_ID msgxA;          /* Message exchange A id      */

union umessage {
  struct cjxmsg umaxmsg;     /* Biggest AMX message        */
  struct usermsg umsg;       /* Define usermsg elsewhere   */
  };


void CJ_CCPP waitmsgxA(void) {
  union umessage msg;        /* Received message           */

                             /* Wait at priority 10        */
                             /* for up to 10 seconds       */
  if (cjmxwait(msgxA, &msg, 10, cjtmconvert(10000))
                  == CJ_EROK) {
         :
         :         /* Process message at msg.umsg          */
         :
                             /* Acknowledge message        */
         cjtkmsgack(CJ_AKBASE + 6);
         }
  }
```

See Also *`cjmxsend, cjtkmsgack`*

Purpose **Build (Create) a Resource Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmbuild(CJ_ID *rmidp,
                           struct cjxsmdef *rmdefp);
```

Description *rmidp* is a pointer to storage for the semaphore id of the resource semaphore allocated to the caller.

rmdefp is a pointer to a resource semaphore definition. Structure *cjxsmdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxsmdef {
  CJ_TAGDEF     xsmdtag;    /* Semaphore tag              */
  int           xsmdvalue;  /* Resource type              */
  };
```

xsmdtag is a 4-character array for the resource semaphore name tag.

xsmdvalue must be one of the following values to specify the type of resource semaphore to be created. The allowable resource types are defined in file *CJZZZSD.H* as follows:

CJ_RMBASIC	Basic resource semaphore
CJ_RMINHERIT	Resource semaphore with priority inheritance

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**rmidp* contains a valid semaphore id.

Errors returned:

For all errors, the semaphore id at **rmidp* is undefined on return.

CJ_ERSMVALUE	Invalid semaphore type (see note below)
CJ_ERSMNONE	No free semaphore

Note The semaphore type must be *CJ_RMBASIC* or *CJ_RMINHERIT*. If value *xsmdvalue >= 0* but *<= 32767 (0x7FFF)*, you will create a counting or bounded semaphore without an error being reported.

...more

Example

```
#include "CJZZZ.H"

static struct cjxsmdef resourcedefA = {
  {"Rm-A"},                   /* Resource semaphore tag    */
  CJ_RMBASIC                  /* Basic resource semaphore  */
  };


CJ_ID CJ_CCPP makeresourceA(void) {
  CJ_ID resourceid;

  if (cjrmbuild(&resourceid, &resourcedefA) == CJ_EROK)
        return(resourceid);
  else   return(CJ_IDNULL); /* Error                     */
  }


CJ_ID CJ_CCPP makeresourceB(void) {
  struct cjxsmdef resourcedefB;
  CJ_ID resourceid;

  *(CJ_TYTAG *)&resourcedefB.xsmdtag = cjcftag("Rm-B");
  resourcedefB.xsmdvalue = CJ_RMINHERIT;

  if (cjrmbuild(&resourceid, &resourcedefB) == CJ_EROK)
        return(resourceid);
  else   return(CJ_IDNULL); /* Error                     */
  }
```

See Also *cjrmcreate, cjrmdelete, cjksfind*

Purpose **Create a Resource Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmcreate(CJ_ID *rmidp, char *tag);
CJ_ERRST CJ_CCPP cjrmcreatex(CJ_ID *rmidp, char *tag, int type);
```

Description Use *cjrmcreate* to create a basic resource semaphore.
Use *cjrmcreatex* to create a basic or priority inheritance resource semaphore.

rmidp is a pointer to storage for the semaphore id of the resource semaphore allocated to the caller.

tag is a pointer to a 4-character string for the resource semaphore name tag.

type must be one of the following values to specify the type of resource semaphore to be created. The allowable resource types are defined in file *CJZZZSD.H* as follows:

CJ_RMBASIC	Basic resource semaphore
CJ_RMINHERIT	Resource semaphore with priority inheritance

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**rmidp* contains a valid semaphore id.

Errors returned:
For all errors, the semaphore id at **rmidp* is undefined on return.

CJ_ERSMVALUE	Invalid semaphore type (see *cjrmbuild* note)
CJ_ERSMNONE	No free semaphore

Example

```
#include "CJZZZ.H"

CJ_ID CJ_CCPP makeresourceC(void) {
  CJ_ID resourceid;

  if (cjrmcreate(&resourceid, "Rm-C") == CJ_EROK)
        return(resourceid);

  else  return(CJ_IDNULL);
  }
```

See Also *cjrmbuild, cjrmdelete, cjksfind*

Purpose	**Delete a Resource Semaphore**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmdelete(CJ_ID resourceid);
```

Description *`resourceid`* is the semaphore id of the resource semaphore to be deleted.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERSMID`	Invalid semaphore id
`CJ_ERSMBUSY`	Resource semaphore is busy One or more tasks are waiting for the resource.

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the resource semaphore. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *`cjrmbuild, cjrmcreate`*

Purpose	**Unconditionally Free a Resource Semaphore**
Used by	■ Task ☐ ISP ☐ Timer Procedure ☐ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmfree(CJ_ID resourceid);
```

Description *resourceid* is the semaphore id of a resource semaphore acquired by a call to *cjrmrsv*.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The resource semaphore nest count is immediately set to zero and the resource is freed. If a priority inheritance resource is being freed and the task priority of the owner has been raised to avoid a priority inversion, the free resource will be granted to the high priority task which is anxiously waiting for the resource. Otherwise, the resource will immediately be given to the task (if any) which is waiting at the head of the resource semaphore wait queue.

Errors returned:

CJ_ERSMID	Invalid semaphore id
CJ_ERSMOWNER	Resource semaphore cannot be released since the calling task does not own the resource.
CJ_ERNOENVLOP	No message envelopes available

Task Switch If the resource is given to a task waiting for the resource, a task switch will occur if the waiting task is of higher priority than the caller.

If a priority inheritance resource is freed by a task whose priority was raised to avoid a priority inversion, that task's priority will be restored upon release of the resource.

Note A task owning more than one priority inheritance resource can release the resources in any order. However, it is recommended that such resources be freed in the opposite order to which they are reserved.

Restrictions You must not attempt to free a counting or bounded semaphore. Use *cjsmsignal* for that purpose.

See Also *cjrmrls, cjrmrsv*

Purpose	**Release a Resource Semaphore**
Used by	■ Task ◻ ISP ◻ Timer Procedure ◻ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmrls(CJ_ID resourceid);
```

Description

resourceid is the semaphore id of a resource semaphore acquired by a call to *cjrmrsv*.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns

Error status is returned.

`CJ_EROK`	Call successful

The resource semaphore nest count is decremented by one. If the nest count goes to zero, the resource is freed. If a priority inheritance resource is being freed and the task priority of the owner has been raised to avoid a priority inversion, the free resource will be granted to the high priority task which is anxiously waiting for the resource. Otherwise, the resource will immediately be given to the task (if any) which is waiting at the head of the resource semaphore wait queue.

If the resource semaphore nest count does not go to zero, the calling task retains ownership of the resource.

Errors returned:

`CJ_ERSMID`	Invalid semaphore id
`CJ_ERSMOWNER`	Resource semaphore cannot be released because the calling task does not own the resource.
`CJ_ERNOENVLOP`	No message envelopes available

Task Switch

If the resource is given to a task waiting for the resource, a task switch will occur if the waiting task is of higher priority than the caller.

If a priority inheritance resource is freed by a task whose priority was raised to avoid a priority inversion, that task's priority will be restored upon release of the resource.

Note

A task owning more than one priority inheritance resource can release the resources in any order. However, it is recommended that such resources be freed in the opposite order to which they are reserved.

Restrictions

You must not attempt to release a counting or bounded semaphore. Use *cjsmsignal* for that purpose.

See Also *cjrmfree, cjrmrsv*

Purpose **Reserve a Resource Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmrsv(CJ_ID resourceid, int priority,
                             CJ_TIME timeout);
```

Description *resourceid* is the semaphore id of the resource semaphore of interest.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*. This parameter is not used if *timeout* is *< 0*.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for the resource. If *timeout = 0*, the caller will wait forever for the resource. If *timeout < 0*, the caller will not be allowed to wait for the resource.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The calling task owns the resource. Upon first acquiring ownership of the resource, the resource semaphore nest count is set to one.

It is permissible for a task which owns the resource to call *cjrmrsv* to reserve the resource again. The resource semaphore nest count is incremented by one for each call to *cjrmrsv* resulting in nested ownership. The resource owner must call *cjrmrls* once for each of its calls to *cjrmrsv*. Alternatively, the resource owner can call *cjrmfree* to unconditionally free the resource.

Warnings returned:

CJ_WRSMINUSE	Resource in use by another task
CJ_WRTMOUT	Timed out before a basic resource became available
CJ_WRTMOUT	The timeout interval has expired. The owner of the priority inheritance resource has not yet released it. However, other activity has allowed the task requesting the resource to resume and detect the timeout.
CJ_WRSMMISS	The priority inheritance resource was released by its owner. However, the timeout interval had expired. Ownership of the resource has been denied because the timeout deadline was missed.

...more

Returns ...continued

Errors returned:

CJ_ERSMID	Invalid semaphore id
CJ_ERSMUV	Resource semaphore usage violation Tasks which own any priority inheritance resources cannot wait for a basic resource semaphore. This error will also occur if tasks are deadlocked or thrashing unsuccessfully trying to resolve a potential priority inversion.

Task Switch If the task waits for a basic resource, there will be an immediate task switch to the next lowest priority ready task.

A task requesting ownership of a priority inheritance resource will never have to wait unless the resource is owned by another lower priority task. Instead, the low priority task will be hoisted to a priority immediately above that of the requesting task, resulting in an immediate task switch to the task which owns the resource. When the resource is eventually released, the priority of the previous owner will be restored and the resource will be granted to the task which has been anxiously awaiting its availability. There will be an immediate task switch to the new owner.

Note A task can own more than one priority inheritance resource. When claiming ownership of multiple resources, tasks competing for the same resources must take care to avoid a deadlock caused by the order in which the requests are made. Although the resources can be released in any order, it is recommended that they be freed in the opposite order to which they are reserved.

Note A task which owns a priority inheritance resource will have its task execution priority automatically raised if any task of higher priority requests ownership of the resource.

Restrictions A task which owns one or more priority inheritance resources must never permit a lower priority task to execute until all of those resources have been freed.

You must not attempt to reserve a counting or bounded semaphore. Use *cjsmwait* for that purpose.

See Also *cjrmfree, cjrmrls*

Purpose **Get Status of a Resource Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjrmstatus(CJ_ID resourceid,
                            struct cjxsmsts *statusp);
```

Description *resourceid* is the semaphore id of the resource semaphore of interest.

statusp is a pointer to storage for the resource semaphore status. Structure *cjxsmsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxsmsts {
   CJ_TYTAG      xsmstag;    /* Semaphore tag             */
   int           xsmsvalue;  /* Semaphore type =          */
                             /* CJ_RMBASIC, CJ_RMINHERIT  */
   int           xsmscount;  /* Semaphore count           */
                             /* 1  free                   */
                             /* 0  owned                  */
                             /* -n owned; n tasks waiting */
   int           xsmsnest;   /* Resource nest level       */
   CJ_ID         xsmsowner;  /* Task id of resource owner */
                             /* if xsmscount <= 0 and     */
                             /* xsmsnest > 0              */
                             /* CJ_IDNULL if resource free*/
                             /* Otherwise, unused         */
   };
```

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the resource semaphore status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERSMID	Invalid semaphore id

Note If the calling task is the resource owner, field *xsmsowner* will always be valid. However, if the calling task is not the resource owner, the owner's task id will only be valid if the resource is owned (*xsmscount <= 0*) and the nesting count is postive (*xsmsnest >= 0*).

Note If field *xsmsvalue >= 0*, the semaphore referenced by *resourceid* is a counting or bounded semaphore. See *cjsmstatus* for an interpretation of the fields in structure *cjxsmsts*.

See Also *cjsmstatus*

This page left blank intentionally.

Purpose **Build (Create) a Counting or Bounded Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmbuild(CJ_ID *smidp,
                           struct cjxsmdef *smdefp);
```

Description *smidp* is a pointer to storage for the semaphore id of the counting or bounded semaphore allocated to the caller.

smdefp is a pointer to a counting or bounded semaphore definition. Structure *cjxsmdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxsmdef {
  CJ_TAGDEF     xsmdtag;    /* Semaphore tag              */
  int           xsmdvalue;  /* Initial semaphore value    */
  };
```

xsmdtag is a 4-character array for the semaphore name tag.

xsmdvalue is the semaphore value which must be specified using one of the following macros defined in file *CJZZZSD.H*.

Use *CJ_SMCOUNT(n)* to create a counting semaphore with an initial value of *n* where *0 <= n <= 16383* (*0x3FFF*) and an absolute upper limit of *16383*.

Use *CJ_SMLIMIT(n)* to create a bounded semaphore with an initial value of *0* and an upper limit of *n* where *0 < n <= 16383*.

Use *CJ_SMBINARY* to create a binary semaphore, a simple bounded semaphore with an initial value of *0* and an upper limit of *1*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**smidp* contains a valid semaphore id.

Errors returned:

For all errors, the semaphore id at **smidp* is undefined on return.

CJ_ERSMVALUE	Invalid semaphore value (see note on next page)
CJ_ERSMNONE	No free semaphore

...more

Note You must use macro `CJ_SMCOUNT(n)`, `CJ_SMLIMIT(n)` or `CJ_SMBINARY` to create a counting or bounded semaphore. Your parameter *n* must be in the range *0* to *16383* (*0x3FFF*).

The macros use your parameter *n* to encode an initial value *v* which includes the semaphore type. Procedures `cjsmbuild` and `cjsmcreate` will return error code `CJ_ERSMVALUE` if the initial value `v < -2`, `v = 0x4000` or `v > 0x7FFF`.

If *v* is `CJ_RMBASIC` (*-1*) or *v* is `CJ_RMINHERIT` (*-2*), a resource semaphore will be created without an error being reported.

Example

```
#include "CJZZZ.H"

static struct cjxsmdef semdefA = {
  {"Sm-A"},                 /* Counting semaphore tag    */
  CJ_SMBINARY               /* Initial semaphore value   */
                            /* for a binary semaphore    */
  };


CJ_ID CJ_CCPP makesemA(void) {
  CJ_ID semid;

  if (cjsmbuild(&semid, &semdefA) == CJ_EROK)
        return(semid);
  else   return(CJ_IDNULL); /* Error                     */
  }


CJ_ID CJ_CCPP makesemB(void) {
  struct cjxsmdef semdefB;
  CJ_ID semid;

  *(CJ_TYTAG *)&semdefB.xsmdtag = cjcftag("Sm-B");
                            /* Counting semaphore with   */
                            /* an initial value of 3     */
  semdefB.xsmdvalue = CJ_SMCOUNT(3);

  if (cjsmbuild(&semid, &semdefB) == CJ_EROK)
        return(semid);
  else   return(CJ_IDNULL); /* Error                     */
  }
```

See Also `cjsmcreate, cjsmdelete, cjksfind`

Purpose **Create a Counting or Bounded Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmcreate(CJ_ID *smidp, char *tag, int value);
```

Description *smidp* is a pointer to storage for the semaphore id of the counting or bounded semaphore allocated to the caller.

tag is a pointer to a 4-character string for the semaphore name tag.

value is the semaphore value which must be specified using one of the following macros defined in file *CJZZZSD.H*.

Use *CJ_SMCOUNT(n)* to create a counting semaphore with an initial value of *n* where *0 <= n <= 16383* (*0x3FFF*) and an absolute upper limit of *16383*.

Use *CJ_SMLIMIT(n)* to create a bounded semaphore with an initial value of *0* and an upper limit of *n* where *0 < n <= 16383*.

Use *CJ_SMBINARY* to create a binary semaphore, a simple bounded semaphore with an initial value of *0* and an upper limit of *1*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**smidp* contains a valid semaphore id.

Errors returned:

For all errors, the semaphore id at **smidp* is undefined on return.

CJ_ERSMVALUE	Invalid semaphore value (see note on previous page)
CJ_ERSMNONE	No free semaphore

Example

```
#include "CJZZZ.H"

CJ_ID CJ_CCPP makesemC(void) {
  CJ_ID semid;

  /* Create a bounded semaphore with an upper limit of 5  */
  if (cjsmcreate(&semid, "Sm-C", CJ_SMLIMIT(5)) == CJ_EROK)
          return(semid);

  else    return(CJ_IDNULL);
  }
```

See Also *cjsmbuild, cjsmdelete, cjksfind*

Purpose **Delete a Counting or Bounded Semaphore**

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

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmdelete(CJ_ID semid);
```

Description *`semid`* is the semaphore id of the counting or bounded semaphore to be deleted.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERSMID`	Invalid semaphore id
`CJ_ERSMBUSY`	Semaphore is busy One or more tasks are waiting for the semaphore.

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the semaphore. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *`cjsmbuild, cjsmcreate`*

Purpose	**Signal to a Counting or Bounded Semaphore**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmsignal(CJ_ID semid);
```

Description

`semid` is the semaphore id of the counting or bounded semaphore of interest.

If a task is waiting for the semaphore, the semaphore is immediately given to the task at the head of the semaphore wait queue. Otherwise, the semaphore value is incremented by one.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns

Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERSMID`	Invalid semaphore id
`CJ_ERSMOVF`	Semaphore value overflow A counting semaphore's value cannot exceed *`16383`*. A bounded semaphore's value cannnot exceed the semaphore's upper limit.
`CJ_ERNOENVLOP`	No message envelopes available

Task Switch

If the semaphore is given to a task waiting for the semaphore, a task switch may occur. If the caller is a task, a task switch will occur if the waiting task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the waiting task is of higher priority than the interrupted task.

Restrictions

You must not attempt to signal to a resource semaphore. Use *`cjrmrls`* or *`cjrmfree`* for that purpose.

See Also *`cjsmwait`*

Purpose **Get Status of a Counting or Bounded Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmstatus(CJ_ID semid,
                            struct cjxsmsts *statusp);
```

Description *semid* is the semaphore id of the counting or bounded semaphore of interest.

statusp is a pointer to storage for the counting or bounded semaphore status. Structure *cjxsmsts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxsmsts {
  CJ_TYTAG       xsmstag;    /* Semaphore tag            */
  int            xsmsvalue;  /* Initial value >= 0       */
                             /* (see note below)         */
  int            xsmscount;  /* Semaphore count          */
                             /* >0 available             */
                             /*  0 not available         */
                             /* -n not available         */
                             /*  n tasks waiting         */
  int            xsmsnest;   /* Always 0                 */
  CJ_ID          xsmsowner;  /* Always CJ_IDNULL         */
  };
```

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the semaphore status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERSMID	Invalid semaphore id

Note If field *xsmsvalue < 0*, the semaphore referenced by *semid* is a resource semaphore. See *cjrmstatus* for an interpretation of the fields in structure *cjxsmsts*.

If field *xsmsvalue < 16384 (0x4000)*, the semaphore referenced by *semid* is a counting semaphore with an initial count of *xsmsvalue*. If field *xsmsvalue > 16384 (0x4000)*, the semaphore is a bounded semaphore with an upper limit of *xsmsvalue & 0x3FFF*.

See Also *cjrmstatus*

Purpose **Wait on a Counting or Bounded Semaphore**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjsmwait(CJ_ID semid, int priority,
                          CJ_TIME timeout);
```

Description *semid* is the semaphore id of the counting or bounded semaphore of interest.

priority is the priority at which the caller wishes to wait (0 = highest). To wait in FIFO order, have all callers use the same value for *priority*. This parameter is not used if *timeout* is < *0*.

timeout > 0 is the maximum interval measured in system ticks which the caller is prepared to wait for the semaphore. If *timeout = 0*, the caller will wait forever for the semaphore. If *timeout < 0*, the caller will not be allowed to wait for the semaphore.

If the semaphore value is > *0*, the value is decremented by one and the calling task is granted use of the semaphore. Otherwise, the calling task is forced to wait for the semaphore (*timeout >= 0*) or is allowed to proceed with a warning (*timeout < 0*).

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful. The calling task can use the semaphore. The semaphore value is decremented by one.

Warnings returned:

CJ_WRSMINUSE	Semaphore not available
CJ_WRTMOUT	Timed out before semaphore became available

Errors returned:

CJ_ERSMID	Invalid semaphore id
CJ_ERSMUV	Resource semaphore usage violation Tasks which own any priority inheritance resources cannot wait for a counting or bounded semaphore.

Task Switch If the task waits for the semaphore, there will be an immediate task switch to the next lowest priority ready task.

Restrictions You must not attempt to wait for a resource semaphore. Use *cjrmrsv* for that purpose.

See Also *cjsmsignal*

Purpose **Format Time and Date as an ASCII String**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
int CJ_CCPP cjtdfmt(struct cjxtd *tdbufp,
                        int format, char *ascii);
```

Description *tdbufp* is a pointer to a time/date structure containing the time and date which is to be formatted. Structure *cjxtd* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtd {
  CJ_T8U      xtdsec;     /* Seconds (0-59)           */
  CJ_T8U      xtdmin;     /* Minutes (0-59)           */
  CJ_T8U      xtdhr;      /* Hours   (0-23)           */
  CJ_T8U      xtdday;     /* Day     (1-31)           */
  CJ_T8U      xtdmonth;   /* Month   (1-12)           */
  CJ_T8U      xtdyear;    /* Year    (0-99)           */
  CJ_T8U      xtdow;      /* Day of week              */
                          /* (Mon=1 to Sun=7)         */
  CJ_T8U      xtdcen;     /* 0 if time/date incorrect */
                          /* Century if time/date     */
                          /* is correct               */
  CJ_ID       xtdid;      /* Time/Date timer id       */
  };
```

format is the format specification value (see Chapter 5.5) consisting of a combination of the following constants:

Pick one of:		**Default** (*format = 0*):	
CJ_TDFTIME	Time only	*"23:59:59 Jan 31/93"*	
CJ_TDFDATE	Date only	If incorrect (*xtdcen = 0*), then	
CJ_TDFT_D	Time & date	*"23:59:59#Jan 31/93"*	
CJ_TDFD_T	Date & time		
Add one of:		**Add any or none of:**	
CJ_TDFmmmDY	*"Jan 31/93"*	*CJ_TDFSSEC*	Suppress seconds
CJ_TDFMDY	*"01/31/93"*	*CJ_TDFDOW*	Show day of week *"Fri 31 Jan/93"*
CJ_TDFDmmmY	*"31 Jan/93"*		
CJ_TDFDMY	*"31/01/93"*	*CJ_TDFCENT*	Show century *"31 Jan/1993"*
CJ_TDFYMD	*"93/01/31"*		

ascii is a 26-byte character array into which the time and/or date will be formatted as a null (*'\0'*) terminated ASCII string.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns The number of characters stored in the *ascii* array, excluding the terminating null (*'\0'*).

See Also *cjtdget*

Purpose **Get the Current Time and Date**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtdget(struct cjxtd *tdbufp);
```

Description *tdbufp* is a pointer to storage for the current time and date. Structure *cjxtd* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtd {
  CJ_T8U        xtdsec;     /* Seconds (0-59)           */
  CJ_T8U        xtdmin;     /* Minutes (0-59)           */
  CJ_T8U        xtdhr;      /* Hours   (0-23)           */
  CJ_T8U        xtdday;     /* Day     (1-31)           */
  CJ_T8U        xtdmonth;   /* Month   (1-12)           */
  CJ_T8U        xtdyear;    /* Year    (0-99)           */
  CJ_T8U        xtdow;      /* Day of week              */
                            /* (Mon=1 to Sun=7)         */
  CJ_T8U        xtdcen;     /* 0 if time/date incorrect */
                            /* Century if time/date     */
                            /* is correct               */
  CJ_ID         xtdid;      /* Time/Date timer id       */
  };
```

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The current AMX time and date are stored in the structure at **tdbufp*.

If field *xtdcen = 0*, the time and date stored at **tdbufp* are valid but are not considered to be the correct time of day or calendar date. Use *cjtdset* to set the correct time and date.

See Also *cjtdfmt, cjtdset*

Purpose **Set the Current Time and Date**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtdset(struct cjxtd *tdbufp);
```

Description *tdbufp* is a pointer to a time/date structure containing the new time and date. Structure *cjxtd* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtd {
  CJ_T8U       xtdsec;     /* Seconds (0-59)            */
  CJ_T8U       xtdmin;     /* Minutes (0-59)            */
  CJ_T8U       xtdhr;      /* Hours   (0-23)            */
  CJ_T8U       xtdday;     /* Day     (1-31)            */
  CJ_T8U       xtdmonth;   /* Month   (1-12)            */
  CJ_T8U       xtdyear;    /* Year    (0-99)            */
  CJ_T8U       xtdow;      /* Day of week               */
                           /* (Mon=1 to Sun=7)          */
  CJ_T8U       xtdcen;     /* 0 if time/date incorrect  */
                           /* Century if time/date      */
                           /* is correct                */
  CJ_ID        xtdid;      /* Time/Date timer id        */
  };
```

If field *xtdcen = 0*, AMX will assume a century of 19. AMX will accept the time and date at **tdbufp* as valid but will not consider the value to be the correct time of day or calendar date. Set field *xtdcen* to the correct century (19, 20, ...) to set the correct time and date.

Fields *xtdow* and *xtdid* are not used by *cjtdset*. The correct day of the week (1 to 7) is automatically computed by *cjtdset* from the calendar date provided.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The AMX system time and date is set to that specified by the caller in the structure at **tdbufp*. The values for time and date supplied by the caller are not checked for validity.

Errors returned:

CJ_ERTMID	Invalid time/date timer id
CJ_ERTMVALUE	Invalid time/date period

These errors indicate that private internal AMX parameters have probably been corrupted.

See Also *cjtdfmt, cjtdget*

Purpose **Build (Create) a New Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkbuild(CJ_ID *tkidp,
                        struct cjxtkdef *tkdefp);
```

Description *tkidp* is a pointer to storage for the task id of the created task.

tkdefp is a pointer to a task definition. Structure *cjxtkdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtkdef {
  CJ_TAGDEF     xtkdtag;     /* Task tag                    */
  CJ_TASKPROC xtkdproc;      /* A(task procedure)           */
  void          *xtkdstore; /* A(bottom of task storage) */
  unsigned int xtkdstsize;   /* Size of task storage        */
                             /* (bytes)                     */
  unsigned int xtkdattr;     /* Task attributes             */
  int           xtkdpr;      /* Task priority               */
  unsigned int xtkdslice;    /* Task time slice             */
                             /* (system ticks)              */
  };
```

xtkdtag is a 4-character array for the task name tag.

xtkdproc is a pointer to the task procedure to be executed whenever the task is triggered by a call to *cjtktrigger*.

xtkdstore is a pointer to the bottom of a statically allocated region of alterable memory (RAM) for use as the task's stack and Task Control Block. The region must be properly aligned for use as a stack. The alignment requirement is dictated by the target processor. For most processors, long alignment is adequate.

xtkdstsize is the size, in bytes, of the task storage referenced by *xtkdstore*. The size must be a multiple of 4.

xtkdattr is reserved to define task attributes. Set *xtkdattr = 0*.

xtkdpr is the priority at which the task is to execute. *Xtkdpr* must be in the range 1 to 127 (1 = highest; 127 = lowest).

xtkdslice is the task's time slice interval measured in system ticks. Set *xtkdslice = 0* if the task is not to be time-sliced.

Interrupts ■ Disabled ☐ Enabled ■ Restored

...more

Returns Error status is returned.

CJ_EROK	Call successful

**tkidp* contains a valid task id.

Errors returned:

For all errors, the task id at **tkidp* is undefined on return.

CJ_ERTKNONE	No free task
CJ_ERTKSTORE	Insufficient storage provided for task stack and Task Control Block
CJ_ERTKPR	Invalid task priority (must be 1 to 127)

Restrictions Once you create a task, it cannot be deleted until the task's termination is enabled using *cjtkterm*.

Example

```
#include "CJZZZ.H"
                             /* Task Procedure A            */
extern void CJ_CCPP taskAproc(void);

                             /* Task Procedure B            */
extern void CJ_CCPP taskBproc(void);

                             /* Task A storage              */
#define STOREA 1024
static long taskAram[STOREA/sizeof(long)];

                             /* Task B storage              */
#define STOREB 2048
static long taskBram[STOREB/sizeof(long)];

static struct cjxtkdef taskdefA = {
  {"Tk-A"},                  /* Task A tag                  */
  (CJ_TASKPROC)taskAproc,    /* Task Procedure A            */
  taskAram, STOREA,          /* Task A storage              */
  0,                         /* Task A attributes           */
  5,                         /* Task A priority             */
  0                          /* Task A is not sliced        */
  };
```

...more

Example ...continued

```
CJ_ID CJ_CCPP maketaskA(void) {
  CJ_ID taskid;

  if (cjtkbuild(&taskid, &taskdefA) == CJ_EROK) {

         /* Start task A                                 */
         if (cjtktrigger(taskid) == CJ_EROK)
                 return(taskid);

         /* Serious fault; fatal error                   */
         cjksfatal(CJ_FEBASE + 1, cjtkid());
         }

  return(CJ_IDNULL);        /* Error                     */
  }


CJ_ID CJ_CCPP maketaskB(void) {
  struct cjxtkdef taskdefB;
  CJ_ID taskid;

  *(CJ_TYTAG *)&taskdefB.xtkdtag = cjcftag("Tk-B");
  taskdefB.xtkdproc = (CJ_TASKPROC)taskBproc;
                            /* Task B storage            */
  taskdefB.xtkdstore = taskBram;
  taskdefB.xtkdstsize = STOREB;
  taskdefB.xtkdattr = 0;    /* Task B attributes         */
  taskdefB.xtkdpr = 10;     /* Task B priority           */
                            /* Task B slice = 100 ms     */
  taskdefB.xtkdslice = cjtmconvert(100);

  if (cjtkbuild(&taskid, &taskdefB) == CJ_EROK) {

         /* Start task B                                 */
         if (cjtktrigger(taskid) == CJ_EROK)
                 return(taskid);

         /* Serious fault; fatal error                   */
         cjksfatal(CJ_FEBASE + 2, cjtkid());
         }

  return(CJ_IDNULL);        /* Error                     */
  }
```

See Also *cjtkcreate, cjtkdelete, cjtkmxinit, cjtkterm, cjksfind*

Purpose **Create a New Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkcreate(CJ_ID *tkidp, char *tag,
                            CJ_TASKPROC proc,
                            void *store, unsigned int stsize,
                            unsigned int attr, int priority,
                            unsigned int slice);
```

Description *tkidp* is a pointer to storage for the task id of the created task.

tag is a pointer to a 4-character string for the task name tag.

proc is a pointer to the task procedure to be executed whenever the task is triggered by a call to *cjtktrigger*.

store is a pointer to the bottom of a statically allocated region of alterable memory (RAM) for use as the task's stack and Task Control Block. The region must be properly aligned for use as a stack. The alignment requirement is dictated by the target processor. For most processors, long alignment is adequate.

stsize is the size, in bytes, of the task storage referenced by *store*. The size must be a multiple of 4.

attr is reserved to define task attributes. Set *attr = 0*.

priority is the priority at which the task is to execute. *Priority* must be in the range 1 to 127 (1 = highest; 127 = lowest).

slice is the task's time slice interval measured in system ticks. Set *slice = 0* if the task is not to be time-sliced.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**tkidp* contains a valid task id.

...more

Returns ...continued

Errors returned:
For all errors, the task id at `*tkidp` is undefined on return.

`CJ_ERTKNONE`	No free task
`CJ_ERTKSTORE`	Insufficient storage provided for task stack and Task Control Block
`CJ_ERTKPR`	Invalid task priority (must be 1 to 127)

Restrictions Once you create a task, it cannot be deleted until the task's termination is enabled using *cjtkterm*.

Example

```
#include "CJZZZ.H"
                                /* Task Procedure C          */
extern void CJ_CCPP taskCproc(void);

                                /* Task C storage            */
#define STOREC 640
static long taskCram[STOREC/sizeof(long)];


CJ_ID CJ_CCPP maketaskC(void) {
  CJ_ID taskid;

  if (cjtkcreate(&taskid,
                   "Tk-C",
                   (CJ_TASKPROC)taskCproc,
                            /* Task C storage            */
                   taskCram, STOREC,
                   0,       /* Task C attributes         */
                   10,      /* Task C priority           */
                            /* Task C slice = 200 ms     */
                   cjtmconvert(200)
                   ) == CJ_EROK) {

          /* Start task C                                */
          if (cjtktrigger(taskid) == CJ_EROK)
                   return(taskid);

          /* Serious fault; fatal error                  */
          cjksfatal(CJ_FEBASE + 3, cjtkid());
          }

  return(CJ_IDNULL);        /* Error                     */
  }
```

See Also *cjtkbuild, cjtkdelete, cjtkmxinit, cjtkterm, cjksfind*

Purpose **Delay for an Interval**

`Cjtkdelay` is similar to `cjtkwaitm` but with the error status codes reversed. For a delay, timeout is expected and a `cjtkwake` request generates a warning. For a timed wait, a timeout generates a warning and a `cjtkwake` request is considered normal.

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkdelay(CJ_TIME interval);
```

Description `interval` is the required delay interval measured in system ticks. `Interval` must be `>= 0`. If `interval = 0`, the calling task will wait forever or until a `cjtkwake` call wakes the task.

Interrupts ■ Disabled ■ Enabled ◻ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Task was delayed for `interval` system ticks.

Warnings returned:

`CJ_WRTKDELAY`	Task wakened by `cjtkwake` before delay completed

Errors returned:

`CJ_ERSMUV`	Resource semaphore usage violation (see `cjtkwait`)

Note If the task has an outstanding `cjtkwake` wake request pending when it calls `cjtkdelay`, the task will continue execution immediately with an error status of `CJ_WRTKDELAY` without any delay.

If there is any possibility that some task, ISP or Timer Procedure has already issued a `cjtkwake` call to wake the task, the task should call `cjtkwaitclr` to reset the pending wake request prior to calling `cjtkdelay`.

Task Switch If the task is allowed to delay, there will be an immediate task switch to the next lowest priority ready task.

See Also `cjtkwait, cjtkwaitclr, cjtkwaitm, cjtkwake`

Purpose **Delete a Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkdelete(CJ_ID taskid, int priority);
```

Description *taskid* is the task id of the task to be deleted. A task can delete itself.

priority is the task execution priority at which the deletion is to occur. *Priority* must be in the range 1 to 127 (1 = highest; 127 = lowest).

The deletion priority must be lower than the priority of any task which can affect the task being deleted. The deletion priority must be higher than the priority of any permanently active compute bound task.

AMX forces the task which is to be deleted to begin (or resume) execution. AMX then changes the task's priority to the deletion priority, calls the task's Task Termination Procedure and frees the task's stack and TCB for reuse.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTKPR	Invalid task priority (must be 1 to 127)
CJ_ERTKABORT	Task cannot be deleted until a call to *cjtkterm* provides a Task Termination Procedure

Note If the task which is being deleted has an unacknowledged message in its possession, AMX will automatically call *cjtkmsgack* to acknowledge the message and return an answer-back status of *CJ_EROK* to the task which is waiting for the acknowledgement.

...more

Task Switch An immediate task switch will occur. AMX invokes the Kernel Task to initiate deletion of the task and then resumes execution of the highest priority ready task.

Restrictions A task cannot be deleted until the task's termination is enabled using *`cjtkterm`*.

You must not use *`cjtkdelete`* to delete a message exchange task. Use *`cjtkxdelete`* for that purpose.

You must not delete a task which is waiting, or is about to wait, for any AMX resource such as a buffer from a buffer pool, a resource or counting semaphore, events in an event group or a message from a mailbox or message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

Example The following example illustrates the steps needed to delete a task and recover the memory used by that task for stack and TCB. You must follow these steps to assure that the task being deleted has truly disappeared before you reuse its stack for some other purpose.

In this example it is assumed that the task which is doing the deletion will find the deleted task's stack by referencing the task definition which was used to create that task (see the example for *`cjtkbuild`*).

...more

Example ...continued

```
#include "CJZZZ.H"

CJ_ERRST CJ_CCPP deltask(
  CJ_ID  taskid,              /* Task id of task to delete */
  int    deletepr)            /* Deletion priority         */
  {
  struct cjxtksts taskinfo;
  CJ_ERRST status;
  int    taskpr;
  CJ_ID  curtask;

                              /* Get task's status         */
  if ((status = cjtkstatus(taskid, &taskinfo)) != CJ_EROK)
        return(status);       /* Error                     */

  taskpr = taskinfo.xtkspr;   /* Save task's priority      */

                              /* Delete the task           */
  if ((status = cjtkdelete(taskid, deletepr)) != CJ_EROK)
        return(status);       /* Error                     */

  curtask = cjtkid();         /* Get current task status   */
  if ((status = cjtkstatus(curtask, &taskinfo)) != CJ_EROK)
        return(status);       /* Error                     */

  if (deletepr > taskpr)      /* Pick lowest priority      */
        taskpr = deletepr;

  if (taskinfo.xtkspr <= taskpr) {
        /* Temporarily drop current task priority       */
        /* below the task being deleted to allow the    */
        /* task deletion to complete                    */
        cjtkpriority(curtask, taskpr + 1);
        cjtkpriority(curtask, taskinfo.xtkspr);
        }

  return (CJ_EROK);
  }
```

See Also *cjtkbuild, cjtkcreate, cjtkterm, cjtkxdelete*

Purpose **End Execution of the Current Task**

A task normally ends when its task procedure returns to AMX. However, under some circumstances, often related to error conditions, a task may wish to end but, because of procedure nesting, cannot easily do so. In such cases the task can call *cjtkend* to force an immediate end to the task.

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

Setup Prototype is in file *CJZZZKF.H*.

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

Description The calling task is immediately forced to end execution.

Interrupts ■ Disabled ■ Enabled □ Restored

Returns There is no return from *cjtkend*.

If the task which is ending has an unacknowledged message in its possession, AMX will automatically call *cjtkmsgack* to acknowledge the message and return an answer-back status of *CJ_EROK* to the task which is waiting for the acknowledgement.

If the task has an outstanding trigger request pending, AMX will immediately call the task procedure again with the task's stack reset for reuse. Otherwise, AMX declares the task idle waiting for a trigger request.

Task Switch If the task becomes idle, there will be an immediate task switch to the next lowest priority ready task.

See Also *cjtktrigger, cjtkmsgack*

Purpose **Get the Task Id of the Current Task**

Reentrant procedures which are shared by several tasks can use *`cjtkid`* to get the task id of the task which is executing the procedure thereby eliminating the need to pass the caller's task id as a parameter to the procedure.

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

Setup Prototype is in file *`CJZZZKF.H`*.

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

Description AMX assures that the highest priority task which is capable of execution has the use of the processor. That task is called the current task. A task which is idle or waiting for something to occur cannot be the current task.

Interrupts □ Disabled □ Enabled □ Restored

Returns When called from a task or an Exit Procedure, the task id of the currently executing task is returned to the caller.

When called from an ISP, the value *`CJ_IDNULL`* is returned to the caller.

See Also *`cjtktcb`*

Purpose **Kill (Flush) a Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkkill(CJ_ID taskid);
```

Description *taskid* is the task id of the task to be killed. A task can kill itself.

AMX forces the task which is to be killed to begin (or resume) execution. AMX then resets the task's trigger count to zero, calls the task's Task Termination Procedure and ends the task.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTKABORT	Task cannot be killed until a call to *cjtkterm* provides a Task Termination Procedure

Note If the task which is being killed has an unacknowledged message in its possession, AMX will automatically call *cjtkmsgack* to acknowledge the message and return an answer-back status of *CJ_EROK* to the task which is waiting for the acknowledgement.

Task Switch An immediate task switch will occur. AMX invokes the Kernel Task to initiate the task kill and then resumes execution of the highest priority ready task.

Restrictions A task cannot be killed until the task's termination is enabled using *cjtkterm*.

You must not use *cjtkkill* to kill a message exchange task. Use *cjtkxkill* for that purpose.

You must not kill a task which is waiting, or is about to wait, for any AMX resource such as a buffer from a buffer pool, a resource or counting semaphore, events in an event group or a message from a mailbox or message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *cjtkterm, cjtkxkill*

Purpose	**Acknowledge Receipt of a Message**
Used by	■ Task ◻ ISP ◻ Timer Procedure ◻ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkmsgack(int ackback);
```

Description *`ackback`* is the answer-back status to be returned to the task which sent the message which the calling task is currently processing. *`Ackback`* must be `>= 0`.

It is recommended that you use status values of *`CJ_AKBASE + n`* where *`n >= 0`* to distinguish your codes from AMX warning codes.

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Warnings returned:

`CJ_WRAKNOTASK`	The message currently being processed did not come from another task
`CJ_WRAKNOWAIT`	No task is waiting for an answer-back to the message currently being processed
`CJ_WRTKWAKEN`	Task not waiting yet and acknowledge already posted (Private AMX data has probably been corrupted causing this warning)

Errors returned:

`CJ_ERAKVALUE`	Answer-back status must be `>= 0`

Task Switch If the task waiting for the message acknowledgement is of higher priority than the calling task, an immediate task switch to that task will occur.

See Also *`cjmbsend, cjmxsend, cjtkend`*

Purpose **Get a Task's Message Exchange Id**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ID CJ_CCPP cjtkmxid(CJ_ID taskid);
```

Description *taskid* is the task id of the message exchange task of interest.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns If *taskid* is not a valid task id, a message exchange id of *CJ_IDNULL* is returned.

If the task identified by *taskid* is a message exchange task (see *cjtkmxinit*), the task's message exchange id is returned.

Otherwise a message exchange id of *CJ_IDNULL* is returned.

Example

```
#include "CJZZZ.H"

CJ_ERRST CJ_CCPP sendMXtask(CJ_ID taskid, void *msgp) {
   CJ_ID mxid;

   /* Find task's message exchange id                         */
   if ( (mxid = cjtkmxid(taskid)) == CJ_IDNULL )
          return(CJ_ERTKID); /* No message exchange           */


   /* Send message to the message exchange task               */
   /* Send at priority 1; do not wait for ACK                 */
   return ( cjmxsend(mxid, msgp, CJ_NO, 1) );
   }
```

See Also *cjtkmxinit*

Purpose **Initialize and Start a Message Exchange Task**

To use this procedure, you must first create a message exchange. Then create a task with a task procedure which expects to receive an AMX message. Finally, call *`cjtkmxinit`* to attach the message exchange to the task and automatically start the task such that it is waiting for a message to arrive at the message exchange. Whenever a message arrives, the task procedure will be called with a copy of the message on its stack ready to be processed.

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

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkmxinit(CJ_ID taskid, CJ_ID mxid);
```

Description *`taskid`* is the task id of the task to which the message exchange is to be attached. The task must be idle.

`mxid` is the message exchange id of the message exchange to be attached to the specified task.

A message exchange task can be created using *`cjtkbuild`* or *`cjtkcreate`*. The task must be created with the task attribute *`CJ_MATAMSG`* indicating that the task will receive an AMX message on its stack.

By default, each AMX message arriving on the task's message exchange is passed to the task procedure by value. Such a task is prototyped as follows:

```
void CJ_CCPP msgxtask(struct cjxmsg message);
```

Alternatively, the AMX messages can be passed to the message exchange task procedure by reference provided the task is created with the additional task attribute *`CJ_MATAPBR`*. Such a task is prototyped as follows:

```
void CJ_CCPP msgxtask(struct cjxmsg *msgp);
```

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTKID`	Invalid task id
`CJ_ERMXID`	Invalid message exchange id

...more

Example

```
#include "CJZZZ.H"
                             /* Task Procedure            */
extern void CJ_CCPP taskproc(struct cjxmsg *msgp);

                             /* Task storage              */
#define STORE 1024
static long taskram[STORE/sizeof(long)];


CJ_ID CJ_CCPP makeMXtask(void) {
  CJ_ID taskid;
  CJ_ID mxid;

  /* Create a message exchange for the task              */
  if (cjmxcreate(&mxid, "T-MX", 10, 10, 0, 0) != CJ_EROK)
         return(CJ_IDNULL); /* Error                      */

  if (cjtkcreate(&taskid,
                   "T-MX",
                   (CJ_TASKPROC)taskproc,
                             /* Task storage              */
                   taskram, STORE,
                             /* Task attributes           */
                   CJ_MATAMSG + CJ_MATAPBR,
                   10,       /* Task priority             */
                   0         /* Task not sliced           */
                   ) == CJ_EROK) {

         /* Start the message exchange task               */
         if (cjtkmxinit(taskid, mxid) == CJ_EROK)
                   return(mxid);

         /* Serious fault; fatal error                    */
         cjksfatal(CJ_FEBASE + 4, cjtkid());
         }

  return(CJ_IDNULL);         /* Error                     */
  }
```

See Also *cjmxcreate, cjtkcreate, cjtkmxid, cjtkxdelete, cjtkxkill*

Purpose	**Sense and/or Adjust a Task's Execution Priority**
Used by	■ Task ◻ ISP ◻ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkpradjust(CJ_ID taskid, int *prp);
```

Description `taskid` is the task id of the task whose priority is to be sensed or changed. A task can sense or change its own priority.

`prp` is a pointer to storage for the task's priority. On entry, `*prp` must be set to specify the required action.

`*prp = -1`	Fetch the task's true priority.
`*prp = 0`	Fetch the task's base priority.
`*prp = n > 0`	Fetch the task's base priority and then change the task's base priority to `n`. Value `n` must be in the range 1 to 127 (1 = highest; 127 = lowest).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

`*prp` is the task's true priority or base priority per the caller's request.
The base priority is the task's normal execution priority, 1 to 127.
The true priority is `(p << 8) + r` where `p` is the priority at which the task is actually executing and `r` is the task's relative priority, from 0 (highest) to 255 (lowest), with respect to all other tasks currently executing at that same priority level `p`.

Errors returned:
`*prp` is undefined.

`CJ_ERTKID`	Invalid task id
`CJ_ERTKPR`	Invalid task priority (`*prp` must be `-1` to `127`)

Note A task normally runs at its base priority, 1 to 127. If a task owns a priority inheritance resource, its true execution priority will be higher than its base priority if AMX has to raise the task priority to avoid a priority inversion.

Task Switch If the task whose priority was changed is ready to run and its new priority is higher than that of the calling task, an immediate task switch to that task will occur.

Restrictions Do not alter the priority of a task which uses priority inheritance resources unless you are certain that the task is not using those resources at the time that you alter the task's priority.

See Also `cjtkcreate, cjtkdelete, cjtkpriority, cjtkstatus`

This page left blank intentionally.

Purpose **Change a Task's Execution Priority**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkpriority(CJ_ID taskid, int priority);
```

Description *taskid* is the task id of the task whose priority is to be changed. A task can change its own priority.

priority is the desired execution priority of the task. *Priority* must be in the range 1 to 127 (1 = highest; 127 = lowest).

Interrupts ■ Disabled ◻ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTKPR	Invalid task priority (must be 1 to 127; see note)

Note Do not use priority 0. Although its use is not prohibited, priority 0 is considered an invalid task priority. Priority 0 is reserved for use by AMX.

You can use procedure *cjtkpradjust* to sense a task's priority without having to call *cjtkstatus* for the task's complete status.

Task Switch If the task whose priority was changed is ready to run and its new priority is higher than that of the calling task, an immediate task switch to that task will occur.

Restrictions Do not alter the priority of a task which uses priority inheritance resources unless you are certain that the task is not using those resources at the time that you alter the task's priority.

Example See the example provided with *cjtkdelete* for one of the few valid uses for this procedure.

See Also *cjtkcreate, cjtkdelete, cjtkpradjust*

Purpose **Resume a Suspended Task**

Resume a task known to be suspended as a result of a *cjtksuspend* call.

There should rarely be a need to call this procedure in a well designed system. *Cjtksuspend* and *cjtkresume* are provided primarily to simplify the porting of designs from other operating systems to AMX.

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkresume(CJ_ID taskid);
```

Description *taskid* is the task id of the task whose suspension is to lifted.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id

Note A task may remain blocked, unable to execute, even though the suspension has been removed. For example, assume that a task which was waiting for a semaphore was subsequently suspended by a *cjtksuspend* call. When the suspension is lifted with a *cjtkresume* call, the task will remain blocked waiting for the semaphore if the semaphore has not yet been granted to the task.

Task Switch If the task whose suspension has been lifted is ready to run, a task switch may occur. If the caller is a task, a task switch will occur if the resumed task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the resumed task is of higher priority than the interrupted task.

See Also *cjtksuspend*

Purpose **Get Status of a Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkstatus(CJ_ID taskid,
                        struct cjxtksts *statusp);
```

Description *taskid* is the task id of the task of interest.

statusp is a pointer to storage for the task status. Structure *cjxtksts* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtksts {
  CJ_TYTAG      xtkstag;    /* Task tag                 */
  CJ_T32U       xtkssts;    /* Task status              */
  CJ_TIME       xtkstmr;    /* Task timer               */
  CJ_TYSLICE    xtksslice;  /* Time slice               */
  int           xtkspr;     /* Task priority            */
  unsigned int xtksattr;    /* Task attributes          */
  int           xtkscalls;  /* Trigger count            */
  CJ_ID         xtksmstid;  /* Message sender task's id */
  CJ_ID         xtksmxid;   /* Task message exchange id */
  };
```

xtkstag is a 4-character array containing the task name tag.

xtkstmr is the number of system ticks remaining in the task's timeout or delay timer. *Xtkstmr = 0* if the task is not in the timed wait state.

xtksslice is the task's time slice interval measured in system ticks. *Xtksslice = 0* if the task is not time-sliced.

xtkspr is the priority at which the task executes.

xtkscalls is task's current trigger count. If *xtkscalls* is not 0, then the task has *xtkscalls* pending requests to execute.

xtksmstid is the task id of the task, if any, which sent the AMX message which the task is currently processing. If the task has not retrieved a message from a mailbox or message exchange or if such a message came from an ISP, *xtksmstid* will be *CJ_IDNULL*.

xtksmxid is the task's message exchange id. If the task is not a message exchange task, *xtksmxid* will be *CJ_IDNULL*.

...more

Description ...continued

xtkssts defines the task state. The task status bits are defined in file *CJZZZSD.H* as follows:

CJ_MATSWTR	Trigger wait (task is idle)
CJ_MATSWSUS	Suspended (waiting for resume)
CJ_MATSWAIT	Waiting (see reasons below)
CJ_MATSWHLT	Halted
	Wait reasons (used with *CJ_MATSWAIT*)
CJ_MATSWSM	Semaphore wait
CJ_MATSWEV	Event group wait
CJ_MATSWMB	Mailbox wait
CJ_MATSWMX	Message exchange wait
CJ_MATSWBUF	Buffer wait
CJ_MATSWTM	Timer wait (may be used with others)
	Extended wait reasons
CJ_MATSWMBAK	Mailbox ACK wait
CJ_MATSWMXAK	Message exchange ACK wait

xtksattr defines the task attributes. The task attribute bits are defined in file *CJZZZSD.H* as follows:

CJ_MATAHLT	Task cannot be halted
CJ_MATAMSG	Message exchange task
CJ_MATAPBR	Task receives message by reference
CJ_MATAMSW	Message sender task is waiting for ACK

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

The structure at **statusp* contains the task status.

Errors returned:

For all errors, the structure at **statusp* is undefined on return.

CJ_ERTKID	Invalid task id

Purpose	**Stop (End) Execution of Task**
Used by	■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkstop(CJ_ID taskid);
```

Description

taskid is the task id of the task to be stopped. A task can stop itself.

AMX forces the task which is to be stopped to begin (or resume) execution. AMX then calls the task's Task Termination Procedure and ends the task.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns

Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTKABORT	Task cannot be stopped until a call to *cjtkterm* provides a Task Termination Procedure

Note

If the task which is being stopped has an unacknowledged message in its possession, AMX will automatically call *cjtkmsgack* to acknowledge the message and return an answer-back status of *CJ_EROK* to the task which is waiting for the acknowledgement.

Task Switch

An immediate task switch will occur. AMX invokes the Kernel Task to initiate the task stop and then resumes execution of the highest priority ready task.

Restrictions

A task cannot be stopped until the task's termination is enabled using *cjtkterm*.

You must not stop a task which is waiting, or is about to wait, for any AMX resource such as a buffer from a buffer pool, a resource or counting semaphore, events in an event group or a message from a mailbox or message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *cjtkterm*

Purpose **Suspend a Task**

There should rarely be a need to call this procedure in a well designed system. `Cjtksuspend` and `cjtkresume` are provided primarily to simplify the porting of designs from other operating systems to AMX.

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtksuspend(CJ_ID taskid);
```

Description `taskid` is the task id of the task to be suspended. A task can suspend itself.

The task will be unconditionally suspended until some other task, ISP or Timer Procedure lifts the suspension with a matching `cjtkresume` call.

Interrupts ■ Disabled □ Enabled ■ Restored

If a task suspends itself, interrupts will be disabled briefly. Upon return, interrupts will be enabled, even if they were disabled upon entry.

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTKID`	Invalid task id

Note A task suspension overrides all other wait conditions. For example, assume that a task which was waiting for a semaphore was subsequently suspended by a `cjtksuspend` call. If the semaphore is granted to the task before the suspension is lifted, the task will remain suspended with the semaphore in its possession waiting for a `cjtkresume` call.

Task Switch If a task suspends itself, there will be an immediate task switch to the next lowest priority ready task.

Restrictions You must not suspend the AMX Kernel Task.

Do not suspend a task which uses priority inheritance resources unless you are certain that the task is not using those resources at the time that you suspend the task.

See Also `cjtkresume`

Purpose **Find a Task's Task Control Block**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtktcb(CJ_ID taskid,
                         struct cjxtcbs **tcbpp);
```

Description *taskid* is the task id of the task of interest.

tcbpp is a pointer to storage for a pointer to the Task Control Block (TCB) of the specified task. Structure *cjxtcbs* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtcbs {
   CJ_T32U xtcbrsv[CJ_MINTCB]; /* Private                  */
   CJ_T32U xtcbuser[4];        /* User defined             */
   };
```

xtcbrsv is private to AMX. Do not reference this array.

xtcbuser is an array of four 32-bit unsigned integers reserved for use by your application.

Interrupts ☐ Disabled ☐ Enabled ☐ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**tcbpp* contains a valid TCB pointer.

Errors returned:

For all errors, the TCB pointer at **tcbpp* is undefined on return.

CJ_ERTKID	Invalid task id

Restrictions Do not use the Task Control Block pointer provided by this procedure if there is any chance that the corresponding task could be deleted while the TCB pointer is in your possession.

Do not make any assumptions about the size of the TCB. The TCB may actually be larger than indicated by the size of structure *cjxtcbs*.

See Also *cjtkid*

Purpose **Enable/Disable Termination of a Task**

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

Setup Prototype is in file `CJZZZKF.H`.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkterm(CJ_ID taskid, CJ_TERMPROC ttproc);
```

Description `taskid` is the task id of the task whose Task Termination Procedure is to be set.

`ttproc` is a pointer to the Task Termination Procedure which is to be executed by the task whenever the task is stopped, killed or deleted.

`ttproc` is prototyped as follows:

```
void CJ_CCPP termfn(CJ_ID taskid, int reason);
```

`taskid` is the task id of the task being terminated.
`reason` is one of the following task termination reasons:

`CJ_KCCFSTOP`	Task is being stopped
`CJ_KCCFKILL`	Task is being killed
`CJ_KCCFDEL`	Task is being deleted

For portability using different C compilers, cast your procedure pointer as `(CJ_TERMPROC)termfn` in your call to `cjtkterm`.

If `ttproc` = `CJ_NULLFN`, termination of the task will be inhibited until a valid Task Termination Procedure is set with `cjtkterm.`

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTKID`	Invalid task id
`CJ_ERTKTERM`	Task termination is in progress

...more

Example

```
#include "CJZZZ.H"

/* Make and start periodic timer B (see cjtmbuild example)*/
extern CJ_ID CJ_CCPP maketimerB(void);

/* Make and trigger task C (see cjtkcreate example)       */
extern CJ_ID CJ_CCPP maketaskC(void);

/* Forward reference to Task C Termination Procedure      */
void CJ_CCPP taskCterm(CJ_ID taskid, int reason);

static CJ_ID taskCid;        /* Task C id                 */


                             /* Task C Restart Procedure  */
void CJ_CCPP taskCrr(void) {

  /* Make task C and trigger it                           */
  /* If task created, allow it to be terminated           */
  if ( (taskCid = maketaskC()) != CJ_IDNULL)
         cjtkterm(taskCid, (CJ_TERMPROC)taskCterm);

  /* Make periodic timer B and start it                   */
  maketimerB();
  }


/* Task C goes compute bound until timer B deletes it     */
void CJ_CCPP taskCproc(void) {
  for (;;) ;
  }


/* Task C Termination Procedure                           */
/* If task being deleted, taskCid is no longer valid      */
void CJ_CCPP taskCterm(CJ_ID taskid, int reason) {
  if (reason == CJ_KCCFDEL)
         taskCid = CJ_IDNULL;
  }


/* Periodic timer B                                       */
/* If task C id is valid, delete task C and stop timer B  */
void CJ_CCPP timerBproc(CJ_ID timerid, void *unused) {
  if (taskCid != CJ_IDNULL) {
         cjtkdelete(taskCid, 10);
         cjtmwrite(timerid, 0);
         }
  }
```

See Also *cjtkdelete, cjtkkill, cjtkstop, cjtkxdelete, cjtkxkill*

Purpose	**Trigger (Start) a Task**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtktrigger(CJ_ID taskid);
```

Description *`taskid`* is the task id of the task to be triggered.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTKID`	Invalid task id

Note AMX starts a task by calling the task procedure identified when the task was created. The task is considered active until the task procedure ends by returning to AMX or by calling *`cjtkend`*.

The task procedure is called by AMX once for each trigger which the task has received. If the task is active when the task is triggered, AMX increments the task's trigger count. When the task ends, AMX checks the task's trigger count and, if it is non zero, decrements the trigger count and calls the task procedure again.

Task Switch If the task trigger is successful, a task switch may occur if the triggered task was idle prior to the trigger. If the caller is a task, a task switch will occur if the triggered task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the triggered task is of higher priority than the interrupted task.

See Also *`cjtkend`*

Purpose **Wait Unconditionally for a Wake Request**

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

Setup Prototype is in file *CJZZZKF.H*.

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

Description The calling task will be suspended until some other task, ISP or Timer Procedure issues a *cjtkwake* call to wake this task.

Interrupts ■ Disabled ■ Enabled □ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Task was wakened by *cjtkwake*.

Errors returned:

CJ_ERSMUV	Resource semaphore usage violation (see note)

Note If the task has an outstanding *cjtkwake* wake request pending when it calls *cjtkwait*, the task will continue execution immediately with status of *CJ_EROK*.

If there is any possibility that some task, ISP or Timer Procedure has already issued a *cjtkwake* call to wake the task, the task should call *cjtkwaitclr* to reset the pending wake request prior to calling *cjtkwait*.

Note A task which owns one or more priority inheritance resources cannot suspend itself for any reason. The task must remain compute bound, executing until it has released all such resources. Then, and only then, can the task wait for some event of interest. Any task which attempts to wait for any reason while owning a priority inheritance resource will resume with error status *CJ_ERSMUV* without waiting.

Task Switch If the task is allowed to wait, there will be an immediate task switch to the next lowest priority ready task.

...more

Example The following example is paired with the example for *cjtkwake*.

```
#include "CJZZZ.H"

extern void deviceon(void);
extern void deviceoff(void);

static CJ_ID taskflag;


void CJ_CCPP devicetask(void) {

  cjtkwaitclr();             /* Reset pending wakes       */
  taskflag = cjtkid();       /* Identify current task     */
  deviceon();                /* Enable device interrupts  */

  for (;;) {
        cjtkwait();          /* Wait for device           */

        if (taskflag == CJ_IDNULL)
                break;       /* Device overrun in ISP     */

        /* Device interrupt detected                      */
        /* Service as necessary                           */

        }                    /* End of for loop           */

  cjtkwaitclr();             /* Reset pending wakes after */
                             /* ISP overrun               */
  }
```

See Also *cjtkdelay, cjtkwaitm, cjtkwaitclr, cjtkwake*

Purpose **Reset Pending Wake Requests**

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

Setup Prototype is in file *CJZZZKF.H*.

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

Description Use *cjtkwaitclr* to reset any pending wakes prior to a call to *cjtkdelay*, *cjtkwait* or *cjtkwaitm*. The pending wakes, if any, usually occur because of errors or race conditions involving calls to *cjtkwake*.

For example, assume that a task calls *cjtkwaitm* to wait for up to 500 ms for an event which will be signalled by a *cjtkwake* call from an ISP. If the interrupt occurs just after the task times out but before the task can take any corrective action, the ISP will issue a *cjtkwake* call which will produce a pending wake request because the task is no longer waiting.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Nothing

Restrictions This procedure can only be called by the currently executing task.

See Also *cjtkdelay, cjtkwait, cjtkwaitm, cjtkwake*

Purpose **Timed Wait for a Wake Request**

Cjtkwaitm is similar to *cjtkdelay* but with the error status codes reversed. For a timed wait, a timeout generates a warning and a *cjtkwake* request is considered normal. For a delay, timeout is expected and a *cjtkwake* request generates a warning.

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkwaitm(CJ_TIME timeout);
```

Description *timeout* is the maximum interval measured in system ticks which the caller is prepared to wait for a *cjtkwake* wake request. *Timeout* must be >= *0*. If *timeout = 0*, the calling task will wait forever or until a *cjtkwake* call wakes the task.

The calling task will be suspended until some other task, ISP or Timer Procedure issues a *cjtkwake* call to wake this task or until the specified time interval expires.

Interrupts ■ Disabled ■ Enabled ◻ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Task was wakened by *cjtkwake* before *timeout* system ticks elapsed.

Warnings returned:

CJ_WRTMOUT	Task timed out with no *cjtkwake* request

Errors returned:

CJ_ERSMUV	Resource semaphore usage violation (see *cjtkwait*)

Note If the task has an outstanding *cjtkwake* wake request pending when it calls *cjtkwaitm*, the task will continue execution immediately with status of *CJ_EROK*.

If there is any possibility that some task, ISP or Timer Procedure has already issued a *cjtkwake* call to wake the task, the task should call *cjtkwaitclr* to reset the pending wake request prior to calling *cjtkwaitm*.

Task Switch If the task is allowed to wait, there will be an immediate task switch to the next lowest priority ready task.

...more

Example The following example is paired with the example for *cjtkwake*.

```
#include "CJZZZ.H"

extern void deviceon(void);
extern void deviceoff(void);

static CJ_ID taskflag;


void CJ_CCPP devicetask(void) {
  CJ_ERRST status;
  CJ_TIME timeout;

                            /* Timeout = 500 ms           */
  timeout = cjtmconvert(500);
  cjtkwaitclr();            /* Reset pending wakes        */
  taskflag = cjtkid();      /* Identify current task      */
  deviceon();               /* Enable device interrupts   */

  for (;;) {
                            /* Wait for device            */
        status = cjtkwaitm(timeout);

        if (status == CJ_WRTMOUT) {
                            /* Disable device interrupts  */
                deviceoff();
                break;
                }

        if (taskflag == CJ_IDNULL)
                break;      /* Device overrun in ISP      */

        /* Device interrupt detected                      */
        /* Service as necessary                           */

        }                   /* End of for loop            */

  cjtkwaitclr();            /* Reset pending wakes after  */
                            /* ISP overrun or timeout     */
  }
```

See Also *cjtkdelay, cjtkwait, cjtkwaitclr, cjtkwake*

Purpose	**Wake a Waiting Task**
	Wake a task known to be waiting because of a *`cjtkwait`* or *`cjtkwaitm`* or call.
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`CJ_ERRST CJ_CCPP cjtkwake(CJ_ID taskid);`

Description	*`taskid`* is the task id of the task waiting for a wake request.
Interrupts	■ Disabled ◻ Enabled ■ Restored
Returns	Error status is returned.

`CJ_EROK`	Call successful

Warnings returned:

`CJ_WRTKWAKEP`	Task not waiting; wake now pending
`CJ_WRTKWAKEN`	Task not waiting; wake already pending

Errors returned:

`CJ_ERTKID`	Invalid task id

Task Switch If the task wake is successful, a task switch may occur. If the caller is a task, a task switch will occur if the waiting task is of higher priority than the caller. If the caller is an ISP, a task switch will occur when the interrupt service is complete if the waiting task is of higher priority than the interrupted task.

...more

Example The following example is paired with the examples for *cjtkwait* and *cjtkwaitm*.

```
#include "CJZZZ.H"

extern void deviceon(void);
extern void deviceoff(void);

extern CJ_ID taskflag;


void CJ_CCPP deviceint(void) {
   CJ_ERRST status;

   /* Device interrupt occurred                           */
   /* Dismiss interrupt and service as necessary          */

   if (taskflag != CJ_IDNULL) {
                            /* Wake the task               */
          status = cjtkwake(taskflag);

          if ( (status == CJ_EROK) ||
               (status == CJ_WRTKWAKEP) )
                   return;  /* All OK                      */
          }

   /* Device overrun detected:                            */
   /*    1. No task available to service device           */
   /* or 2. Task already had a pending wake request       */

   taskflag = CJ_IDNULL;    /* Tell task ISP overran     */
   deviceoff();             /* Disable device interrupts */
   }
```

See Also *cjtkdelay, cjtkwait, cjtkwaitm*

Purpose **Delete a Message Exchange Task**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkxdelete(CJ_ID taskid, int priority);
```

Description *taskid* is the task id of the message exchange task to be deleted. A message exchange task can delete itself.

priority is the task execution priority at which the deletion is to occur. *Priority* must be in the range 1 to 127 (1 = highest; 127 = lowest).

The deletion priority must be lower than the priority of any task which can affect the task being deleted. The deletion priority must be higher than the priority of any permanently active compute bound task.

AMX forces the task which is to be deleted to begin (or resume) execution. AMX then changes the task's priority to the deletion priority, flushes all messages from the task's message exchange and deletes the message exchange. Finally, AMX calls the task's Task Termination Procedure and frees the task's stack and TCB for reuse.

Interrupts ■ Disabled ■ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTKMX	Task has no message exchange
CJ_ERTKPR	Invalid task priority (must be 1 to 127)
CJ_ERTKABORT	Task cannot be deleted until a call to *cjtkterm* provides a Task Termination Procedure

Note If the task which is being deleted has an unacknowledged message in its possession, AMX will automatically call *cjtkmsgack* to acknowledge the message and return an answer-back status of *CJ_EROK* to the task which is waiting for the acknowledgement.

The task's message exchange is flushed as described for procedure *cjmxflush*.

...more

Task Switch An immediate task switch will occur. AMX invokes the Kernel Task to initiate deletion of the task and then resumes execution of the highest priority ready task.

Restrictions A task cannot be deleted until the task's termination is enabled using *`cjtkterm`*.

You must only use use *`cjtkxdelete`* to delete a message exchange task. Use *`cjtkdelete`* to delete all other tasks.

You must not delete a task which is waiting, or is about to wait, for any AMX resource such as a buffer from a buffer pool, a resource or counting semaphore, events in an event group or a message from a mailbox or message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *`cjmxflush, cjmxdelete, cjtkdelete, cjtkmxinit, cjtkterm`*

Purpose	**Kill (Flush) a Message Exchange Task**
Used by	■ Task ■ ISP ■ Timer Procedure □ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtkxkill(CJ_ID taskid);
```

Description

`taskid` is the task id of the message exchange task to be killed. A message exchange task can kill itself.

AMX forces the message exchange task which is to be killed to begin (or resume) execution. AMX then resets the task's trigger count to zero, flushes all messages from the task's message exchange, calls the task's Task Termination Procedure and ends the task.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns

Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTKID`	Invalid task id
`CJ_ERTKMX`	Task has no message exchange
`CJ_ERTKABORT`	Task cannot be killed until a call to *`cjtkterm`* provides a Task Termination Procedure

Note

If the task which is being killed has an unacknowledged message in its possession, AMX will automatically call *`cjtkmsgack`* to acknowledge the message and return an answer-back status of *`CJ_EROK`* to the task which is waiting for the acknowledgement.

The task's message exchange is flushed as described for procedure *`cjmxflush`*.

...more

Task Switch An immediate task switch will occur. AMX invokes the Kernel Task to initiate the task kill and then resumes execution of the highest priority ready task.

Restrictions A task cannot be killed until the task's termination is enabled using `cjtkterm`.

You must only use use `cjtkxkill` to kill a message exchange task. Use `cjtkkill` to kill all other tasks.

You must not kill a task which is waiting, or is about to wait, for any AMX resource such as a buffer from a buffer pool, a resource or counting semaphore, events in an event group or a message from a mailbox or message exchange. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also `cjmxflush, cjtkkill, cjtkterm`

Purpose	**Build (Create) an Interval Timer**
Used by	■ Task ☐ ISP ☐ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup

Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmbuild(CJ_ID *tmidp,
                           struct cjxtmdef *tmdefp);
```

Description

tmidp is a pointer to storage for the timer id of the interval timer allocated to the caller.

tmdefp is a pointer to an interval timer definition. Structure *cjxtmdef* is defined in file *CJZZZSD.H* as follows:

```
struct cjxtmdef {
  CJ_TAGDEF      xtmdtag;    /* Timer tag                  */
  CJ_TMRPROC     xtmdproc;   /* Timer Procedure pointer    */
  CJ_TIME        xtmdperiod; /* Timer period               */
  void           *xtmdparam; /* A(Timer parameter)         */
  };
```

xtmdtag is a 4-character array for the interval timer name tag.

xtmdproc is a pointer to the Timer Procedure to be executed whenever the timer expires.

xtmdperiod is the timer interval, measured in system ticks, to be used if the timer is periodic. *Xtmdperiod* must be *>= 0*. Set *xtmdperiod = 0* for a one-shot timer.

xtmdparam is a pointer to an application specific parameter or variable. The pointer *xtmdparam* is passed to the Timer Procedure whenever it is called by AMX. If no parameter is required by the Timer Procedure, set *xtmdparam = NULL*.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns

Error status is returned.

CJ_EROK	Call successful

**tmidp* contains a valid timer id.

Errors returned:

For all errors, the timer id at **tmidp* is undefined on return.

CJ_ERTMVALUE	Invalid timer period (must be *>= 0*)
CJ_ERTMNONE	No free timer

...more

Note Creating an interval timer does not automatically start the timer. Use `cjtmwrite` to start the timer.

Example

```
#include "CJZZZ.H"
extern long timerAinfo;        /* Timer A parameter          */
                               /* Timer Procedure A          */
extern void CJ_CCPP timerAproc(CJ_ID timerid, long *paramAp);
                               /* Timer Procedure B          */
extern void CJ_CCPP timerBproc(CJ_ID timerid, void *unused);

static struct cjxtmdef timerdefA = {
  {"Tm-A"},                    /* Timer A tag                */
  (CJ_TMRPROC)timerAproc,      /* Timer Procedure A          */
  0,                           /* Timer A is not periodic    */
  &timerAinfo                  /* A(timer A parameter)       */
  };

CJ_ID CJ_CCPP maketimerA(void) {
  CJ_ID timerid;

  if (cjtmbuild(&timerid, &timerdefA) == CJ_EROK) {
        /* Start timer A                                     */
        if (cjtmwrite(timerid, 1) == CJ_EROK)
                return(timerid);

        /* Serious fault; delete timer A                     */
        cjtmdelete(timerid);
        }
  return(CJ_IDNULL);           /* Error                      */
  }

CJ_ID CJ_CCPP maketimerB(void) {
  struct cjxtmdef timerdefB;
  CJ_ID timerid;

  *(CJ_TYTAG *)&timerdefB.xtmdtag = cjcftag("Tm-B");
  timerdefB.xtmdproc = (CJ_TMRPROC)timerBproc;
  timerdefB.xtmdperiod = cjtmconvert(1000); /* 1 second   */
  timerdefB.xtmdparam = NULL;

  if (cjtmbuild(&timerid, &timerdefB) == CJ_EROK) {
        /* Start timer B                                     */
        if (cjtmwrite(timerid, timerdefB.xtmdperiod)
                           == CJ_EROK)
                return(timerid);

        /* Serious fault; delete timer B                     */
        cjtmdelete(timerid);
        }
  return(CJ_IDNULL);           /* Error                      */
  }
```

See Also `cjtmcreate, cjtmdelete, cjksfind`

Purpose **Convert Milliseconds to System Ticks**

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

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_TIME CJ_CCPP cjtmconvert(unsigned long ms);
```

Description *`ms`* is the number of milliseconds.

Interrupts □ Disabled □ Enabled □ Restored

Returns *`ntick`* = the number of system ticks which will occur in *`ms`* milliseconds.

If the clock frequency is such that *`ms`* milliseconds requires more than 2^{31} system ticks, *`ntick`* will be set to 2^{31}-1.

If the clock frequency is such that *`ms`* milliseconds is less than one half of a system tick, *`ntick`* will be set to 0.

Restrictions Avoid using this procedure in ISPs or Timer Procedures where execution speed is critical. For example, if an ISP must start a timer, use *`cjtmconvert`* during initialization to derive *`ntick`*. The ISP can then use *`ntick`* to start the timer without the execution penalty imposed by *`cjtmconvert`*.

Purpose **Create an Interval Timer**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmcreate(CJ_ID *tmidp, char *tag,
                        CJ_TMRPROC tmproc, CJ_TIME tmperiod,
                        void *tmparam);
```

Description *tmidp* is a pointer to storage for the timer id of the interval timer allocated to the caller.

tag is a pointer to a 4-character string for the interval timer name tag.

tmproc is a pointer to the Timer Procedure to be executed whenever the timer expires.

tmperiod is the timer interval, measured in system ticks, to be used if the timer is periodic. *Tmperiod* must be >= *0*. Set *tmperiod = 0* for a one-shot timer.

tmparam is a pointer to an application specific parameter or variable. The pointer *tmparam* is passed to the Timer Procedure whenever it is called by AMX. If no parameter is required by the Timer Procedure, set *tmparam = NULL*.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**tmidp* contains a valid timer id.

Errors returned:

For all errors, the timer id at **tmidp* is undefined on return.

CJ_ERTMVALUE	Invalid timer period (must be >= *0*)
CJ_ERTMNONE	No free timer

Note Creating an interval timer does not automatically start the timer. Use *cjtmwrite* to start the timer.

...more

Example

```
#include "CJZZZ.H"
                             /* Timer C parameter          */
extern struct timerdata timerCinfo;

                             /* Timer Procedure C          */
extern void CJ_CCPP timerCproc(CJ_ID timerid,
                             struct timerdata *tmrCdatap);


CJ_ID CJ_CCPP maketimerC(void) {
  CJ_ID timerid;

  if (cjtmcreate(&timerid,
                 "Tm-C",
                 (CJ_TMRPROC)timerCproc,
                 0, &timerCinfo) == CJ_EROK) {

         /* Start timer C                                   */
         if (cjtmwrite(timerid, 5) == CJ_EROK)
                 return(timerid);

         /* Serious fault; delete timer C                   */
         cjtmdelete(timerid);
         }

  return(CJ_IDNULL);        /* Error                        */
  }
```

See Also *cjtmbuild, cjtmdelete, cjksfind*

Purpose	**Delete an Interval Timer**
Used by	■ Task ☐ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmdelete(CJ_ID timerid);
```

Description *`timerid`* is the timer id of the interval timer to be deleted.

Interrupts ■ Disabled ☐ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTMID`	Invalid timer id
`CJ_ERTMBUSY`	Interval timer is active

Note A Timer Procedure can delete the interval timer with which it is associated.

An active interval timer cannot be deleted. Use *`cjtmwrite`* to stop the timer before you try to delete it.

Restrictions You must be absolutely certain that no other task, ISP or Timer Procedure is in any way using or about to use the interval timer. Failure to observe this restriction may lead to unexpected and unpredictable faults.

See Also *`cjtmbuild, cjtmcreate, cjtmwrite`*

Purpose **Read the Time Remaining in an Interval Timer**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmread(CJ_ID timerid, CJ_TIME *tvalp);
```

Description *timerid* is the timer id of the interval timer of interest.

tvalp is a pointer to storage for the current value of the timer measured in system ticks.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

CJ_EROK	Call successful

**tvalp* contains the timer value measured in system ticks. The value is the number of system ticks remaining before the timer will expire.

Errors returned:

For all errors, the value at **tvalp* is undefined on return.

CJ_ERTMID	Invalid timer id

See Also *cjtmwrite*

Purpose **Change a Task's Time Slice Interval**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmslice(CJ_ID taskid, unsigned int tslice);
```

Description *taskid* is the task id of the task whose time slice interval is to be altered.

tslice is the new time slice interval measured in system ticks.
(*0 <= tslice < 65536*)

Set *tslice = 0* to stop time slicing the task.

Interrupts □ Disabled □ Enabled □ Restored

Returns Error status is returned.

CJ_EROK	Call successful

Errors returned:

CJ_ERTKID	Invalid task id
CJ_ERTMVALUE	Invalid time slice interval

See Also *cjtmtsopt*

Purpose **Read System Tick Counter**

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

Setup Prototype is in file *CJZZZKF.H*.

```
#include "CJZZZ.H"
CJ_T32U CJ_CCPP cjtmtick(void);
```

Description AMX maintains a 32-bit unsigned count of the number of elapsed system ticks. The counter wraps from 2^{32}-1 to 0.

The AMX system tick counter will match the hardware clock tick count only if the AMX system clock is configured such that every hardware clock tick is an AMX system tick.

Interrupts □ Disabled □ Enabled □ Restored

Returns The current value of the 32-bit unsigned AMX system tick counter.

Purpose	**Enable or Disable Time Slicing**
Used by	■ Task ■ ISP ■ Timer Procedure ■ Restart Procedure ■ Exit Procedure

Setup	Prototype is in file *`CJZZZKF.H`*.
	`#include "CJZZZ.H"`
	`void CJ_CCPP cjtmtsopt(int option);`

Description	*`option = CJ_NO`* to disable time slicing. *`option = CJ_YES`* to enable time slicing.
Interrupts	■ Disabled □ Enabled ■ Restored
Returns	Nothing
See Also	*`cjtmslice`*

Purpose **Start/Stop an Interval Timer**

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

Setup Prototype is in file *`CJZZZKF.H`*.

```
#include "CJZZZ.H"
CJ_ERRST CJ_CCPP cjtmwrite(CJ_ID timerid, CJ_TIME tval);
```

Description *`timerid`* is the timer id of the interval timer of interest.

`tval` is the timer interval measured in system ticks. *`Tval`* must be >= *`0`*. Set *`tval = 0`* to stop the timer.

Interrupts ■ Disabled □ Enabled ■ Restored

Returns Error status is returned.

`CJ_EROK`	Call successful

Errors returned:

`CJ_ERTMID`	Invalid timer id
`CJ_ERTMVALUE`	Invalid interval value (must be >= *`0`*)

Note A Timer Procedure associated with a periodic timer can stop the timer by calling *`cjtmwrite`* with *`tval = 0`*. The periodic timer can be restarted by calling *`cjtmwrite`* with *`tval = n`*. The timer will expire after *`n`* system ticks and resume periodic operation with its predefined period.

Non periodic timers (one-shots) can be retriggered by their associated Timer Procedure using *`cjtmwrite`* to restart the timer.

See Also *`cjtmread`*

A. AMX Reserved Words

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*
CJZZZFFF.XXX	AMX filenames
CJZZZ.H	Generic AMX include file

The *ZZZ* in each AMX filename is the 3-digit AMX part number used by KADAK to identify the AMX target processor. For example, file *CJZZZSD.H* is an AMX header file for any version of AMX. File *CJ999SD.H* is the same AMX header file for the version of AMX which operates on the processor identified by KADAK part number *999*.

File *CJZZZ.H* is a generic include file which includes the subset of target specific AMX header files needed for compilation of your application C code. By including file *CJZZZ.H* in your source modules, your AMX application becomes readily portable to other target processors.

The generic include file *CJZZZ.H* is a copy of the corresponding part numbered AMX file. For example, if you are developing for the processor identified by KADAK part number *999*, the file *CJZZZ.H* is a copy file *CJ999.H*.

Symbol	**Type**	**File**	**Purpose**
CJ_CCPP	*#define*	*CJ_ZZZCC.H*	C parameter passing keyword
CJ_CCISIZE	*#define*	*CJ_ZZZCC.H*	Size of integer
CJ_CCONST1	*#define*	*CJ_ZZZCC.H*	*const* precedes type
CJ_CCONST2	*#define*	*CJ_ZZZCC.H*	*const* follows type
CJ_CCHUGE	*#define*	*CJ_ZZZCC.H*	Use *huge* for large arrays
CJ_CCREG	*#define*	*CJ_ZZZCC.H*	C supports *register* variables
CJ_CCVOL	*#define*	*CJ_ZZZCC.H*	C supports *volatile* variables
CJ_MAXMSZ	*#define*	*CJ_ZZZSD.H*	Default AMX message envelope size
CJ_MAXMSZ	*#define*	*CJ_ZZZAPP.H*	AMX message envelope size as revised by your application

Symbol	Type	File	Purpose
CJ_IDNULL	*#define*	*CJ_ZZZCC.H*	Null AMX object id (handle)
CJ_NULL	*#define*	*CJ_ZZZCC.H*	Null data or data pointer
CJ_NULLFN	*#define*	*CJ_ZZZCC.H*	Null function pointer
CJ_YES	*#define*	*CJ_ZZZCC.H*	Boolean true
CJ_NO	*#define*	*CJ_ZZZCC.H*	Boolean false
CJ_PROCID	*#define*	*CJ_ZZZCC.H*	Processor identifier
CJ_ID	*typedef*	*CJ_ZZZCC.H*	AMX object id (handle)
CJ_TIME	*typedef*	*CJ_ZZZCC.H*	Interval timer value
CJ_T8	*typedef*	*CJ_ZZZCC.H*	Signed 8-bit value
CJ_T16	*typedef*	*CJ_ZZZCC.H*	Signed 16-bit value
CJ_T32	*typedef*	*CJ_ZZZCC.H*	Signed 32-bit value
CJ_T8U	*typedef*	*CJ_ZZZCC.H*	Unsigned 8-bit value
CJ_T16U	*typedef*	*CJ_ZZZCC.H*	Unsigned 16-bit value
CJ_T32U	*typedef*	*CJ_ZZZCC.H*	Unsigned 32-bit value
CJ_TMRPROC	*typedef*	*CJ_ZZZCC.H*	Timer Procedure pointer
CJ_TASKPROC	*typedef*	*CJ_ZZZCC.H*	Task Procedure pointer
CJ_ISPPROC	*typedef*	*CJ_ZZZCC.H*	ISP pointer
CJ_VOIDPROC	*typedef*	*CJ_ZZZCC.H*	Void function pointer *void (*)(void)*
CJ_VPPROC	*typedef*	*CJ_ZZZCC.H*	Void function pointer *void (*)(void *)*
CJ_ERRST	*#define*	*CJ_ZZZKC.H*	AMX error status
CJ_MAXxxxxx	*#define*	*CJ_ZZZKC.H*	Maximum parameter value
CJ_MINxxxxx	*#define*	*CJ_ZZZKC.H*	Minimum parameter value
CJ_TYTAG	*typedef*	*CJ_ZZZKC.H*	Tag value
CJ_TAGDEF	*typedef*	*CJ_ZZZKC.H*	Tag definition
CJ_PRVNxxxx	*#define*	*CJ_ZZZKT.H*	Processor vector numbers
CJ_MAXxxxxx	*#define*	*CJ_ZZZKT.H*	Maximum parameter value
CJ_MINxxxxx	*#define*	*CJ_ZZZKT.H*	Minimum parameter value
CJ_TRAPPROC	*typedef*	*CJ_ZZZKT.H*	Task Trap Handler pointer
CJ_TYREG	*typedef*	*CJ_ZZZKT.H*	Processor register
CJ_TYFLAGS	*typedef*	*CJ_ZZZKT.H*	Processor flags register
CJ_EVxxxxxx	*#define*	*CJ_ZZZSD.H*	Event Manager constants
CJ_TDFxxxxx	*#define*	*CJ_ZZZSD.H*	Time/Date format specifications
CJ_MATSxxxx	*#define*	*CJ_ZZZSD.H*	Task status masks
CJ_MATAxxxx	*#define*	*CJ_ZZZSD.H*	Task attribute masks
CJ_MAFNxxxx	*#define*	*CJ_ZZZSD.H*	Scheduler Hook masks
CJ_KCCFxxxx	*#define*	*CJ_ZZZSD.H*	Task Termination Procedure reason codes
CJ_TERMPROC	*typedef*	*CJ_ZZZSD.H*	Task Termination Procedure pointer
CJ_TDPROC	*typedef*	*CJ_ZZZSD.H*	Time/Date Scheduling Procedure pointer

B. AMX Error Codes

AMX error codes are signed integers. Codes less than zero are error codes. Codes greater than zero are warning codes. To assist you during testing, the hexadecimal value of the least significant 16-bits of the error code is listed as it might appear in a register or memory dump.

Mnemonic	**Value (dec)**	**Value (hex)**	**Meaning**
CJ_EROK	*0*	*0*	Call successful
CJ_ERTKID	*-1*	*0xFFFF*	Invalid task id
CJ_ERTKNONE	*-2*	*0xFFFE*	No free task
CJ_ERTKPR	*-3*	*0xFFFD*	Invalid task priority
CJ_ERTKSTORE	*-4*	*0xFFFC*	Task storage too small
CJ_ERTKABORT	*-5*	*0xFFFB*	Task abort not allowed
CJ_ERTKTERM	*-6*	*0xFFFA*	Task termination in progress
CJ_ERTKTRAP	*-7*	*0xFFF9*	Invalid task trap
CJ_ERTKMX	*-8*	*0xFFF8*	Task message exchange mismatch
CJ_ERTMID	*-10*	*0xFFF6*	Invalid timer id
CJ_ERTMNONE	*-11*	*0xFFF5*	No free interval timer
CJ_ERTMBUSY	*-12*	*0xFFF4*	Timer busy (cannot delete)
CJ_ERTMVALUE	*-13*	*0xFFF3*	Invalid timing interval
CJ_ERNOENVLOP	*-15*	*0xFFF1*	No message envelope available
CJ_ERNOEXIST	*-16*	*0xFFF0*	Cannot find id of object with tag/key
CJ_ERNOACCESS	*-17*	*0xFFEF*	Cannot access interrupt vector
CJ_ERRANGE	*-18*	*0xFFEE*	Out of range
CJ_ERFORMAT	*-19*	*0xFFED*	Format error (function dependent)
CJ_ERMBID	*-20*	*0xFFEC*	Invalid mailbox id
CJ_ERMBNONE	*-21*	*0xFFEB*	No free mailbox
CJ_ERMBBUSY	*-22*	*0xFFEA*	Mailbox busy (cannot delete)
CJ_ERMBFULL	*-23*	*0xFFE9*	Mailbox full
CJ_ERMBDEPTH	*-24*	*0xFFE8*	Invalid mailbox depth
CJ_ERAKNOTASK	*-27*	*0xFFE5*	Message not from another task (Conversion interface only)
CJ_ERAKVALUE	*-28*	*0xFFE4*	Answer-back status must be >= 0
CJ_ERAKNEED	*-29*	*0xFFE3*	Cannot wait for another message until current message has been ACKed
CJ_ERSMID	*-30*	*0xFFE2*	Invalid semaphore id
CJ_ERSMNONE	*-31*	*0xFFE1*	No free semaphore
CJ_ERSMBUSY	*-32*	*0xFFE0*	Semaphore busy (cannot delete)
CJ_ERSMVALUE	*-33*	*0xFFDF*	Invalid semaphore value
CJ_ERSMOWNER	*-34*	*0xFFDE*	Only owner can release a resource
CJ_ERSMOVF	*-35*	*0xFFDD*	Semaphore count overflow
CJ_ERSMUV	*-36*	*0xFFDC*	Semaphore usage violation

AMX Error Codes (continued)

Mnemonic	Value (dec)	Value (hex)	Meaning
CJ_EREVID	*-40*	*0xFFD8*	Invalid event group id
CJ_EREVNONE	*-41*	*0xFFD7*	No free event group
CJ_EREVBUSY	*-42*	*0xFFD6*	Event group busy (cannot delete)
CJ_ERBMID	*-50*	*0xFFCE*	Invalid buffer pool id
CJ_ERBMNONE	*-51*	*0xFFCD*	No free buffer pool
CJ_ERBMBUSY	*-52*	*0xFFCC*	Pool busy (cannot delete)
CJ_ERBMSIZE	*-53*	*0xFFCB*	Buffer size too small
CJ_ERBMNBUF	*-54*	*0xFFCA*	No buffers provided
CJ_ERBMNOUSE	*-55*	*0xFFC9*	Buffer not in use
CJ_ERBMUSEOVF	*-56*	*0xFFC8*	Buffer use count overflow
CJ_ERBMBADP	*-57*	*0xFFC7*	Invalid buffer pointer
CJ_ERMMID	*-60*	*0xFFC4*	Invalid memory pool id
CJ_ERMMNONE	*-61*	*0xFFC3*	No free memory pool
CJ_ERMMSIZE	*-62*	*0xFFC2*	Memory size too small
CJ_ERMMBADP	*-63*	*0xFFC1*	Invalid memory block pointer
CJ_ERMMNOUSE	*-64*	*0xFFC0*	Memory block not in use
CJ_ERMMUSEOVF	*-65*	*0xFFBF*	Memory block use count overflow
CJ_ERMMGROW	*-66*	*0xFFBE*	Cannot grow size of block (resize)
CJ_ERMMALIGN	*-67*	*0xFFBD*	Memory not long aligned
CJ_ERMXID	*-70*	*0xFFBA*	Invalid message exchange id
CJ_ERMXNONE	*-71*	*0xFFB9*	No free message exchange
CJ_ERMXBUSY	*-72*	*0xFFB8*	Message exchange busy (cannot delete)
CJ_ERMXFULL	*-73*	*0xFFB7*	Message exchange full
CJ_ERMXDEPTH	*-74*	*0xFFB6*	Invalid exchange mailbox depth
CJ_ERMXMBNUM	*-75*	*0xFFB5*	Invalid exchange mailbox number

AMX Warning Codes

Mnemonic	Value (dec)	Value (hex)	Meaning
CJ_WRTKWAKEP	*1*	*0x0001*	Task not waiting; 1 wake pending
CJ_WRTKWAKEN	*2*	*0x0002*	Task not waiting; >1 wake pending
CJ_WRTKDELAY	*3*	*0x0003*	Task wake occurred during delay
CJ_WRTMOUT	*4*	*0x0004*	Timed out
CJ_WRAKNOWAIT	*5*	*0x0005*	Message sender is not waiting for ACK
CJ_WRMBEMPTY	*6*	*0x0006*	Mailbox is empty
CJ_WRSMINUSE	*7*	*0x0007*	Semaphore is in use
CJ_WRBMNOBUF	*8*	*0x0008*	No buffer available
CJ_WRBMMEMSIZ	*9*	*0x0009*	Not enough memory for n buffers
CJ_WRMMNOMEM	*10*	*0x000A*	No memory available
CJ_WRMXEMPTY	*11*	*0x000B*	Message exchange is empty
CJ_WREVNOEVT	*12*	*0x000C*	No event match
CJ_WRMBFLUSH	*13*	*0x000D*	Mailbox flushed
CJ_WRMXFLUSH	*14*	*0x000E*	Message exchange flushed
CJ_WRTKFLUSH	*15*	*0x000F*	Task flushed
CJ_WRAKNOTASK	*16*	*0x0010*	Message not from another task
CJ_WRSMMISS	*17*	*0x0011*	Semaphore deadline missed

AMX Fatal Exit Codes

Mnemonic	Value (dec)	Value (hex)	Meaning
CJ_FENOMEM	*1*	*0x0001*	Not enough Kernel Data memory
CJ_FETRAP	*2*	*0x0002*	Fatal exception trap
CJ_FEISPTRAP	*3*	*0x0003*	Task exception trap in ISP
CJ_FETKTRAP	*4*	*0x0004*	Task exception trap occurred: in a Restart Procedure or in a Timer Procedure or in a task with no task trap handler
CJ_FENOEXIT	*5*	*0x0005*	No exit from permanent system
CJ_FEROMSYS	*6*	*0x0006*	ROMed Kernel received a request which it is not configured to handle
CJ_FEBPNEST	*7*	*0x0007*	Nested breakpoint encountered
CJ_FEBPISP	*8*	*0x0008*	Cannot breakpoint on ISP
CJ_FEBPVECT	*9*	*0x0009*	Breakpoint Manager cannot install vectors
CJ_FECFG	*10*	*0x000A*	Invalid System Configuration
CJ_FENOIRB	*11*	*0x000B*	No Interrupt Request Blocks available
CJ_FESMUV	*12*	*0x000C*	Semaphore usage violation

This page left blank intentionally.

C. Configuration Generator Specifications

C.1 Introduction

If you are not doing your software development on a PC or compatible running Microsoft® Windows®, then you will be unable to use the interactive Configuration Manager for Windows to create and edit your AMX System Configuration Module. You may, however, still be able to use the Configuration Generator to assist you in this process by porting it to your development system as described in Appendix C.4.

As described in Chapter 15.2, the Configuration Generator merges the information from your User Parameter File with a template of a standard System Configuration Module to produce your module. Since the User Parameter File is a text file, you are free to use the text editor of your choice to create and/or edit this file.

The System Configuration Template provided with AMX has been coded in C. Although a portable subset of the C language has been used, you may have to edit the template file to reflect the capabilities of your particular C compiler.

For these reasons, source code of the template file and the Configuration Generator program has been provided with AMX to allow the Configuration Generator to be ported to your software development environment.

To assist you in this process, the specifications for the User Parameter File and the System Configuration Template are presented in this appendix.

C.2 User Parameter File Specification

The User Parameter File is a text file structured as illustrated in Figure C.2-1. The file consists of a sequence of keywords of the form `...XXX` which begin in column one. Each keyword is followed by one or more parameters which you must provide.

```
; Constant definitions
...UPT      NKG,UMS,CLP,CLF,NTK,NTM
...MGR      NMB,NSM,NEV,NBP,NMP,NMX
...OPT      TSLICE,ROMS,CAT,BKPT
...STK      KSS,ISS
;
; Time/Date Manager selection
...TAD      MYSHED
;
; Your task definitions
...TDT      PROC,TKTAG,ATTR,STORE,PR,SLICE,TKID
;
; Your message exchange task definitions
...TMX      QD0,QD1,QD2,QD3,TKMXTAG,TKID
;
; Your Restart Procedures
...RRX      RRPROC1
;
; Your Exit Procedures
...EXX      EXPROC1
;
; Your interval timer definitions
...TMR      PROC,PERIOD,PARAM,TMTAG,TMID
;
; Your semaphore definitions
...SEM      SMVAL,SMTAG,SMID
;
; Your event group definitions
...EVG      EVAL,EVTAG,EVID
;
; Your mailbox definitions
...MBX      QDEPTH,MBTAG,MBID
;
; Your message exchange definitions
...MEX      QDO,QD1,QD2,QD3,MXTAG,MXID
;
; Your buffer pool definitions
...PDT      BPN,BPS,BPTAG,BPID
;
; Your memory pool definitions
...MEM      MPP,MPS,MPTAG,MPID,MPTYPE
```

Figure C.2-1 User Parameter File

The example in Figure C.2-1 uses symbolic names for all of the parameters following each of the keywords. The symbols correspond to the screen fields described in Chapter 15. You are referred to that chapter for detailed descriptions of each of the parameters.

The order of keywords in the User Parameter File is not particularly critical. For convenience, the keywords have been ordered to closely follow the order of the corresponding entries in your System Configuration Module.

The file begins with a set of **constant definitions**.

`NKG`	Number of message envelopes
`UMS`	Size of AMX message
`CLP`	AMX clock period (in hardware ticks)
`CLF`	Hardware clock frequency (hz)
`NTK`	Maximum number of tasks
`NTM`	Maximum number of timers
`NMB`	Maximum number of mailboxes
`NSM`	Maximum number of semaphores
`NEV`	Maximum number of event groups
`NBP`	Maximum number of buffer pools
`NMP`	Maximum number of memory pools
`NMX`	Maximum number of message exchanges
`TSLICE`	Time slice option (0 = No; 1 = Yes)
`ROMS`	AMX is installed in separate ROM (0 = No; 1 = Yes)
`CAT`	AMX configuration attributes (set to 0)
`BKPT`	AMX Breakpoint Manager used (0 = No; 1 = Yes)
`KSS`	AMX Kernel Stack size
`ISS`	AMX Interrupt Stack size

The **Time/Date Manager** is selected as follows. If you have provided a Time/Date Scheduling Procedure, include its name as illustrated. If you do not have such a procedure, omit the name but keep the line with keyword *`...TAD`*.

If you do not want to use the Time/Date Manager, delete the line with keyword *`...TAD`*.

Each of your predefined **tasks** must be defined using keyword *...TDT*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any tasks, delete the line with keyword *...TDT.*

The parameters in each task definition are as follows.

PROC	Task procedure name
TKTAG	Task tag
ATTR	Attributes
STORE	Task storage size (bytes)
PR	Task priority
SLICE	Time slice interval
TKID	Task id variable name

Note that the task tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **message exchange tasks** must be defined using keyword *...TMX*. The order of these definitions, although not critical, will determine the order in which AMX will initialize and start these tasks. For each line with keyword *...TMX* there must be a corresponding and matching task definition line with keyword *...TDT*. If you do not wish to predefine any message exchange tasks, delete the line with keyword *...TMX.*

For each message exchange task, the definitions describe the private message exchange for the task. The parameters in each private message exchange definition are as follows.

QD0	Mailbox 0 depth
QD1	Mailbox 1 depth
QD2	Mailbox 2 depth
QD3	Mailbox 3 depth
TKMXTAG	Task message exchange tag
TKID	Task id variable name

Note that the task tag must include exactly four displayable characters. Spaces and tabs are not allowed. The task message exchange tag is usually chosen to match the task's tag. The task's message exchange id will be stored in id variable *TKID_mx*.

Define all of your **Restart Procedures** in the order in which you wish them to be executed. Note that you are only defining your own procedures. Restart Procedures required by any of the AMX Managers will automatically be installed for you.

Define all of your **Exit Procedures** in the order in which you wish them to be executed. If you have no Exit Procedures, omit the line containing keyword *...EXX*.

Each of your predefined **interval timers** must be defined using the keyword *...TMR*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any timers, delete the line with keyword *...TMR*.

The parameters in each timer definition are as follows:

PROC	Timer Procedure name
PERIOD	Timer period in system ticks (0 = oneshot)
PARAM	Parameter variable name
TMTAG	Timer tag
TMID	Timer id variable name

Note that the timer tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **semaphores** must be defined using the keyword *...SEM*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any semaphores, delete the line with keyword *...SEM*.

The parameters in each semaphore definition are as follows:

SMVAL	Initial semaphore value - 1 for resource semaphore >=0 for counting semaphore
SMTAG	Semaphore tag
SMID	Semaphore id variable name

Note that the semaphore tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **event groups** must be defined using the keyword *...EVG.* The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any event groups, delete the line with keyword *...EVG*.

The parameters in each event group definition are as follows:

EVAL	Initial value of event flags
EVTAG	Event group tag
EVID	Event group id variable name

Note that the event group tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **mailboxes** must be defined using the keyword *...MBX*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any mailboxes, delete the line with keyword *...MBX*.

The parameters in each mailbox definition are as follows:

QDEPTH	Message queue depth
MBTAG	Mailbox tag
MBID	Mailbox id variable name

Note that the mailbox tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **message exchanges** must be defined using the keyword *...MEX*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any message exchanges, delete the line with keyword *...MEX*.

The parameters in each message exchange definition are as follows:

QD0	Message queue 0 depth
QD1	Message queue 1 depth
QD2	Message queue 2 depth
QD3	Message queue 3 depth
MXTAG	Message exchange tag
MXID	Message exchange id variable name

Note that the message exchange tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **buffer pools** must be defined using the keyword *...PDT*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any buffer pools, delete the line with keyword *...PDT*.

The parameters in each buffer pool definition are as follows:

BPN	Number of buffers in the buffer pool
BPS	Size in bytes of each buffer
BPTAG	Buffer pool tag
BPID	Buffer pool id variable name

Note that the buffer pool tag must include exactly four displayable characters. Spaces and tabs are not allowed.

Each of your predefined **memory pools** must be defined using the keyword *`...MEM`*. The order of these definitions will determine their order of creation by AMX. If you do not wish to predefine any memory pools, delete the line with keyword *`...MEM`*.

The parameters in each memory pool definition are as follows:

`MPP`	Memory pool pointer
`MPS`	Size in bytes of the memory pool
`MPTAG`	Memory pool tag
`MPID`	Memory pool id variable name
`MPTYP`	Memory pool pointer type
	0 = absolute address
	1 = external array of *`MPS`* characters

Note that the memory pool tag must include exactly four displayable characters. Spaces and tabs are not allowed.

C.3 System Configuration Template

The System Configuration Template is a source file which defines a System Configuration Module for any system using AMX and its managers. It is recommended that you list file *CJZZZCG.CT* and examine it carefully before trying to read this description.

The template file consists of KADAK macro definitions for all of the various components of a System Configuration Module. The macro definitions are followed by code generating instances of these macros. Conditional assembly is used to inhibit elements in the module which are not required if the corresponding AMX option has not been selected.

A KADAK macro consists of a macro definition statement, a body of source language statements and directives and a macro end statement. The macro definition and end statements are a special pair of directives defined by KADAK.

A simple example is shown below.

```
[~...MAC    ...TMR      PROC,PERIOD,PARAM,TMTAG,TMID
                                    /* Timer Procedure          */
void CJ_CCPP ~1~(CJ_ID timerid, void *tmparam);

#if (~P3~)
extern void * ~3~;                  /* Timer parameter          */
#endif
[~...ENDM
```

The macro definition statement contains a macro directive (*[~...MAC*), a parameter keyword (*...TMR*) and an optional comment string (*PROC,PERIOD,etc.*). These three fields are separated by one or more space and/or tab characters.

The macro directive must start in the first column. The first character of the directive must be the KADAK macro identification character *[*.

The second character in the directive is the parameter delimiter character. This can be any printable character but is reserved exclusively for identifying parameters within the body of the macro. It must not appear in any other context within the macro body. We have chosen the character ~ because it is not required in any of the macro definitions in the System Configuration Template.

Immediately after the parameter delimiter character comes the macro definition keyword *...MAC*.

The parameter keyword (*...TMR*) identifies a particular keyword in the User Parameter File which contains the parameter list needed for expansion of the macro body.

The comment string is optional and is normally used to provide a descriptive list of the macro parameters. The comment string is not used in the parameter substitution process.

Macro parameters appear in the macro body as parameter identifiers. A parameter identifier is a decimal integer preceded and followed by the delimiter character defined in the macro definition statement (*~1~*, *~2~*, etc.). The number identifies the parameter in the particular parameter list specified by the parameter keyword. The parameter is copied from the User Parameter File into the source language macro body replacing the particular identifier. The parameters are numbered 1 to n from left to right in the parameter list following the keyword in the User Parameter File.

A second form of macro parameter has the letter *P* preceding the number in the parameter identifier, as in the *#if* statement in the example. This form directs the Configuration Generator to replace the parameter identifier with the character 1 if the identified parameter is present in the parameter list. If the parameter is missing from the parameter list, the Configuration Generator replaces the parameter identifier with the character 0.

In the above example the macro requires the first and third parameter values from a line in the User Parameter File containing the keyword *...TMR*. If the third parameter is missing from the keyword parameter list in the User Parameter File, the *#if* statement will be false, inhibiting the compiler directive *extern* which, without valid arguments, would produce a compilation error.

The macro is terminated by the macro end directive containing the macro keyword *...ENDM*.

When the Configuration Generator encounters a macro definition in the System Configuration Template File, it scans the User Parameter File for a matching parameter keyword. The macro body is expanded and copied into the System Configuration Module File once for each matching parameter keyword located in the User Parameter File.

A third type of directive is used to define an incremental integer variable. An incremental variable is defined with an initial value and an increment which is added to the variable each time a macro containing the variable is expanded by the Configuration Generator. Note that this *[~...VAR* directive begins with the KADAK macro identification character *[* and a parameter delimiter character as required by all KADAK macro directives. The variable name is a string of printable characters preceded and followed by one or more space and/or tab characters. Although only one incremental variable is allowed, its name, initial value and increment can be redefined and used with other macros.

The example below illustrates the use of an incremental variable.

```
[~...VAR    &TN 1,1     Start at task 1
[~...MAC    ...TDT      PROC,TKTAG,ATTR,STORE,PR,SLICE,TKID
                        /* Storage for task &TN (tag ~2~) */
static CJ_HUGE char cj_kdstore&TN[~4~];
[~...ENDM
```

The first statement in this example is a directive defining an incremental variable named *&TN*. Both its initial value and increment are 1.

Each time the Configuration Generator expands the macro, the current value of the incremental variable is substituted for the variable name wherever it appears within the macro body. For example, assume that the fifth task definition in the User Parameter File is listed as follows:

```
...TDT       taskproc5,TSK5,0,2048,0,task5id
```

When the Configuration Generator encounters this task definition, the current value of the incremental variable `&TN` will be 5. Therefore the body of the macro will be expanded as follows:

```
                        /* Storage for task 5 (tag TSK5) */
static CJ_HUGE char cj_kdstore5[2048];
```

Note that the value of the incremental variable is not updated by its increment until after the macro end directive is encountered.

To gain a better insight to this whole process, run the Configuration Manager on a PC even if you are not doing development on a PC. Use it to create a simple AMX system with two or three predefined tasks. Use the manager to generate the System Configuration Module. Then view the User Parameter File and the System Configuration Module produced by the Configuration Manager. Compare the latter file with the System Configuration Template to see exactly how the parameter identifiers have been replaced by parameters from your User Parameter File.

C.4 Porting the Configuration Generator

The Configuration Manager uses the Configuration Generator to generate your System Configuration Module. If you are not doing your development on a PC or compatible, you may wish to port the Configuration Generator to your development system.

The Configuration Generator is a utility program provided with AMX. The program is written in C with portability in mind. Source file *`CJZZZCG.C`* is delivered with AMX.

To port the Configuration Generator to your development system, compile the module *`CJZZZCG.C`* using your C compiler. Link the resulting object module with your C Run-Time Library to produce a version of the Configuration Generator which will execute in your environment.

The command line syntax to run the Configuration Generator is as follows.

```
CJZZZCG upfname ctfname scfname -q -w -bsss -esss
```

upfname	User Parameter File filename
ctfname	System Configuration Template filename
scfname	System Configuration Module filename

All three filenames must be present in the order specified. Each must include the full path to the file if the file is not in the current directory. Each filename must include both name and extension.

Command line switches beginning with `-` are optional and must follow the filenames.

The *`-q`* (quiet) switch, if present, will inhibit the display of all informational or error messages on the standard output device (*`stdout`*). This switch is used by the Configuration Manager to inhibit the Configuration Generator from writing to the display screen when it is invoked by the manager.

The Configuration Generator will ignore a KADAK macro in the template file if the User Parameter File contains no instances of the macro keyword. By default, there is no indication in the output file that the unused macro has been stripped. If the *`-w`* switch is used, the Configuration Generator will echo the macro definition statement to the output file and follow it with the comment "*`Unused in this configuration`*".

By default, the Configuration Generator uses the macro identification character *[* from the macro directive as a comment character to identify the comments inserted into the output file in response to the *-w* switch.

This default assumption can be overridden using the *-b* and *-e* switches to define alternate comment delimiter strings. The strings *sss* can be up to 16 characters long. Special characters must be inserted using the character pair *^x* for each special character. Special characters are:

^b	blank (space)	(*ASCII 0x20*)
^t	tab	(*ASCII 0x09*)
^r	return	(*ASCII 0x0D*)
^n	new-line	(*ASCII 0x0A*)
*^^*	the character ^	(*ASCII 0x5E*)

Since the System Configuration Module file *scfname* will be a C source file, you should always use the switches *-b/** and *-e*/* with the *-w* switch to force the Configuration Generator's comment strings to begin and end with the C comment delimiter strings */** and **/*.

D. AMX Library Construction

D.1 Building the AMX Library

AMX Libraries are delivered prebuilt, thoroughly exercised and ready for use with the toolsets supported by KADAK. There should be no need to remake the AMX Library although some developers do so just to confirm that they can. Obviously, if you have altered the AMX source code, you will have to rebuild the AMX Library. In some cases, you may be using an out-of-date toolset which requires a rebuild of AMX for backwards compatibility.

If you rebuild AMX with a new set of tools that has not yet been tested by KADAK, it is possible that you will encounter toolset warnings or errors. For example, the tool vendor may have changed the syntax for some assembly language directives or revised the definition of its archive or link or locate specification files. In the worst case, the vendor may have introduced a code generation fault which produces an invalid code sequence within the AMX kernel. If you suspect such a fault, revert to the AMX Library shipped with AMX and report the problem to KADAK's technical support staff.

Let *`nnn`* be the three digit AMX part number identifying the particular variant of AMX that you wish to rebuild. See the list of AMX part numbers in Chapter 1.2 of the Getting Started manual.

Let *`XX`* be the two or three character toolset id for the toolset combination for which you wish to remake AMX. See the list of supported toolset ids in Chapter 1.5 of the Getting Started manual.

It is assumed that AMX has been installed into directory `...\AMXnnn` ready for use with toolset *`XX`*.

Environment Variables

Before constructing the AMX Library, you must first set several Windows environment variables which are needed to build AMX. Be sure to check the list of required Windows environment variables presented in the header of toolset dependent batch file `ENnnnXX.BAT` in AMX directory `...\AMXnnn\MAKE`.

Environment variables `PATH`, `TMP` and `TEMP` are required by most toolsets. Variable `PATH` allows the command line tools to be run. Variables `TMP` and/or `TEMP` provide access to a directory which the tools can use for temporary file storage.

Environment variables `TOOLPATH` and `CJPATH` are required by the AMX batch files used to run the tools when making the AMX Library. Variable `TOOLPATH` provides the full path to the directory in which the tools were installed. Variable `CJPATH` provides the full path to the AMX installation directory `...\AMXnnn`.

Environment variable `AMX_ENDN` is also required to rebuild AMX. Set variable `AMX_ENDN=L` to make little endian libraries. Set `AMX_ENDN=B` to make big endian libraries. The variable will have no effect if the endian nature of the processor is fixed.

Other toolset dependent environment variables such as `AMX_TSAFE` may also be required to rebuild AMX. Such variables, if required, are documented in the header of the toolset dependent batch file `ENnnnXX.BAT` in AMX directory `...\AMXnnn\MAKE`.

Making the AMX Library

To construct the AMX Library, you must first open a Windows Command Prompt window. From the Windows Start, Run... menu, type *cmd* and press Enter. Alternatively, select Command Prompt from the Windows Start menu or any of its sub-menus.

Make sure that you have set all of the Windows environment variables as described on the previous page.

Make AMX directory *...\AMXnnn\MAKE* the current directory and run toolset dependent batch file *ENnnnXX.BAT* in that directory. You must do this to specify the version number of the toolset *XX* tools that you are using to rebuild AMX. In most cases, the version number for the C/C++ compiler is used to indicate the toolset version.

In the following example, the toolset version number is given the name *toolver* for illustration purposes. Use the value which corresponds to the version of the tools which you are using. Allowable values are documented in the header of the toolset dependent batch file *ENnnnXX.BAT*.

```
...\AMXnnn\MAKE>ENnnnXX toolver
```

Within AMX directory *...\AMXnnn\MAKE* you will find two other toolset dependent files which are used to build the AMX Library.

File *CJnnnXX.MAK* is the make specification file which will be used by your *MAKE* utility to build the library. This file purposely avoids constructs and directives that tend to vary among make utilities. You may have to edit this file to meet the requirements of your particular make utility.

File *CJnnnXX.BAT* is a batch file used by the make specification file to run the command line tools for toolset *XX*.

To build AMX, you will need a make utility such as Microsoft *NMAKE.EXE*. Your Windows *PATH* environment variable must provide access to this utility. Before starting the make process, delete all header files (**.DEF* and **.H*), if any, which previous builds may have left in directory *...\AMXnnn\MAKE*. Then issue the following command.

```
! Make AMX using Microsoft NMAKE
...>NMAKE -fCJnnnXX.MAK "TOOLSET=XX"
```

When the make is complete, directory *..\TOOLXX\LIB* will contain your updated AMX Library and object modules.

Directory *..\ERR* will contain zero or more text files which summarize the error messages, if any, produced during the make process.

Note that if you add the Microsoft */N* switch immediately following the *NMAKE* directive on the command line, the make utility will list the make operations on the screen but will not actually do the make. This can be helpful in locating path problems if you have not properly installed AMX or have not provided correct Windows environment variables.

Common Problems

When rebuilding the AMX Library, a number of Windows related or *MAKE* dependent problems may be encountered.

Your *MAKE* utility must be able to issue simple commands such as *COPY* and *ERASE*. It must also be able to invoke the batch file *CJnnnXX.BAT* which is used to run the assembler, C compiler, linker/locator and librarian for toolset *XX*.

Some *MAKE* utilities are provided in different forms for use in different environments. Choose the simplest version which can be executed within the Windows Command Prompt window. Note that the AMX make process has only been tested in a standard Windows Command Prompt environment. The AMX make process is not intended for use under stand-alone DOS or MS-DOS.

So what is a standard Windows Command Prompt environment? It is the configuration of Windows on your PC which allows your *MAKE* utility to run batch file *CJnnnXX.BAT* and still have enough memory free to use the assembler, C/C++ compiler, linker/locator and librarian for toolset *XX*. Try to start with at least 500K of memory available.

RAM drives and temporary disk storage can also be problems. If all of your extended memory is used for a RAM drive, there may not be enough memory free for use by the *MAKE* utility and the software tools. If the drive specified by your *TMP* or *TEMP* environment variable for use for temporary files is almost full, compilations or links may fail. In the past, some tools have been observed to crash or hang if they run out of memory or disk space. Sad but true!

In some cases, the Windows command line length can impose a restriction which may affect the construction process if your *CJPATH* environment variable specifies a long path string. For example, if you install AMX 68000 in directory *D:\PROJECT\YOURAPP*, then variable *CJPATH* must be defined as *CJPATH=D:\PROJECT\YOURAPP\AMX532*. The build process will then use *CJPATH* to reference source and object files like:

```
D:\PROJECT\YOURAPP\AMX532\TOOLXX\SRC\CJ532KA.C
D:\PROJECT\YOURAPP\AMX532\TOOLXX\LIB\CJ532KA.O
```

If both of these strings appear in a single command within the batch file *CJnnnXX.BAT*, the command might fail.

If you suspect that this problem is occurring, use the *SUBST* command to substitute a single drive letter, say *Z:*, for the path string *D:\PROJECT\YOURAPP* as in the following example.

```
    SUBST Z: D:\PROJECT\YOURAPP
    SET CJPATH=Z:
```

This page left blank intentionally.

D.2 A Custom AMX Library

AMX Libraries are delivered prebuilt, thoroughly exercised and ready for use with the toolsets supported by KADAK. There should be no need to remake the AMX Library. However, there are a few adaptations that can be made to reconfigure the AMX Library to meet your special needs.

To implement any of these adaptations, you must make minor edits to AMX header source file *CJnnnAPP.H* in AMX installation directory *AMXnnn\TOOLXX\DEF*.

Altering the AMX Message Size

AMX messages originate as user defined blocks of 12 or more sequential bytes of memory. The maximum length (*n >= 12*) is determined by you when you create your System Configuration Module. If you declare the message length to be greater than 12, you must edit AMX header source file *CJnnnAPP.H* and define symbol *CJ_MAXMSZ* to have the value specified by your configuration.

Symbol *CJ_MAXMSZ* is normally not defined in header file *CJnnnAPP.H*. Its definition is required to ensure that your application modules and your AMX System Configuration Module use the configured AMX message length. If the message length specified in your system configuration does not match that specified in header file *CJnnnAPP.H*, you will get a compilation error when you try to compile your AMX System Configuration Module.

Although you must edit file *CJnnnAPP.H* to change the AMX message length, there is no need to rebuild the AMX Library.

Using Resource Manager Functions

The AMX Semaphore Manager supports all forms of resource semaphore. To distinguish the use of resource semaphores from all other kinds of semaphores, the resource management procedures are given names of the form *cjrmXXXX*.

By default, the AMX resource management procedures are mapped directly to procedures within the AMX Semaphore Manager. This mapping may complicate the testing of your AMX system since the resource procedures will not appear in your symbol table and will therefore not be accessible by your debugger. To overcome this difficulty, edit AMX header source file *CJnnnAPP.H* and define symbol *CJ_OPTRM* as follows:

```
#define CJ_OPTRM
```

The mere definition of symbol *CJ_OPTRM* ensures that your application modules and your AMX System Configuration Module will reference the resource procedures in the AMX Library instead of mapping them to equivalent procedures in the AMX Semaphore Manager.

Although you must edit file *CJnnnAPP.H* to add your definition of symbol *CJ_OPTRM*, there is no need to rebuild the AMX Library.

Reducing AMX Code Size

Many of the functions within the AMX Library include parameter checking sequences upon entry to the functions to detect receipt of invalid parameters from your application code. These checks reject invalid calls, returning the appropriate error code to the caller.

By default, AMX funnels all errors and warnings which it generates through to the AMX User Error Procedure *cjkserror* in source file *CJnnnUF.C* as described in Chapter 14.2.

You can make a minor reduction in the AMX the code footprint and improve the execution speed of some functions by revising the AMX Library to omit these parameter checks and error traps. Edit AMX header source file *CJnnnAPP.H* and define symbol *CJ_OPTVAL*, *CJ_OPTERROR* or *CJ_OPTWARN* to be *0* (instead of *1*) to omit the corresponding feature.

If you edit file *CJnnnAPP.H* and revise the definitions of any of the above symbols, you must rebuild the AMX Library as described in Appendix D.1.

Omitting Priority Inversion Detection

The AMX Semaphore Manager supports resource semaphores which use priority inheritance to resolve priority inversions. This feature was added to the AMX kernel in early 2003. You may wish to rebuild the AMX Library to omit this feature so that your AMX Library resembles that of an earlier AMX release, albeit with all other revisions in place. Doing so will reduce the code footprint of the AMX Semaphore Manager but will have little impact on the speed of execution of its services.

Symbol *CJ_OPTINHERIT* in AMX header source file *CJnnnAPP.H* controls support for priority inheritance as a method for resolving priority inversions within resource semaphores which permit such resolution.

To omit this feature, edit file *CJnnnAPP.H* and define symbol *CJ_OPTINHERIT* to be *0*, instead of its default value of *n = 1000*.

The magic number *n* is an upper bound used by the Semaphore Manager to prevent endless thrashing by tasks which are deadlocked or otherwise hopelessly tangled in their attempts to resolve multiple, simultaneous priority inversions. If a task has to raise the priority of a lower priority task more than *n* times in order to resolve a priority inversion involving a specific resource, then the tasks are probably deadlocked for other reasons.

Reducing the value of the magic number *n* to some value less than *5* may permit you to detect occurrences of complex priority inversion scenarios which, although eventually resolved successfully, are actually adversely affecting the timing of events within your application. The value 1000 was arbitrarily chosen to ensure that serious deadlocks arising from inversions are eventually reported to your application.

If you edit file *CJnnnAPP.H* and alter the value of symbol *CJ_OPTINHERIT*, you must rebuild the AMX Library as described in Appendix D.1.