

# **Getting Started**

**with the**

## **AMX<sup>™</sup> Multitasking Executive**

**First Printing: June 16, 1993**

**Last Printing: March 1, 2005**

**Copyright © 1993 - 2005**

**KADAK Products Ltd.**

**206 - 1847 West Broadway Avenue**

**Vancouver, BC, Canada, V6J 1Y5**

**Phone: (604) 734-2796**

**Fax: (604) 734-8114**



## TECHNICAL SUPPORT

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

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

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

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

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

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

**Copyright © 1993-2005 by KADAK Products Ltd.  
All rights reserved.**

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

### **DISCLAIMER**

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

### **TRADEMARKS**

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

# GETTING STARTED WITH AMX

## Table of Contents

	Page
<b>1. Welcome to AMX</b>	<b>1</b>
1.1 Introduction to AMX .....	1
1.2 Installing the AMX Software .....	2
1.3 Installing the AMX Prototyping System (TAPS) .....	5
1.4 Installing the <i>KwikLook</i> Fault Finder .....	7
1.5 Choosing Your Toolset .....	9
1.6 Program Groups .....	11
1.7 Uninstalling AMX, TAPS or <i>KwikLook</i> .....	12
<b>2. AMX Sample Program</b>	<b>13</b>
2.1 Sample Program Operation .....	13
2.2 Building the AMX Sample Program .....	20
2.3 Using a Toolset IDE .....	24
2.4 The TAPS Sample Program .....	25
<b>3. Creating an AMX Application</b>	<b>27</b>
3.1 The AMX Configuration Process .....	27
3.2 Building Your AMX Application .....	31
3.3 A Make File for AMX Applications .....	34
3.4 Using the AMX Prototyping System (TAPS) .....	36
<b>4. Debugging an AMX Application</b>	<b>39</b>
4.1 Using the <i>KwikLook</i> Fault Finder .....	39
4.2 Breakpoints and Tracing .....	42
4.3 Debugging the Launch .....	43
4.4 Debugging Caveats .....	45
<b>5. AMX Programming Hints</b>	<b>47</b>
5.1 Application Portability .....	47
5.2 AMX Stack Allocation .....	48
5.3 Choosing a Synchronization Method .....	50
5.4 AMX Caveats .....	53
5.5 Interrupt Latency .....	56
<b>6. C Programming Primer</b>	<b>57</b>
6.1 C Programming Practices .....	57
6.2 Structure Packing .....	59
6.3 Reentrancy and Concurrent Execution .....	60
6.4 Using the C Runtime Library .....	62
6.5 Bootstrap and C Startup Code .....	64

This page left blank intentionally.

# 1. Welcome to AMX

## 1.1 Introduction to AMX

Welcome to real-time software development with the AMX™ Multitasking Executive.

Chapter 1 of this "Getting Started" guide will lead you through the installation process and provide you with a roadmap to the installed AMX software components and online documentation.

Chapter 2 describes a simple AMX Sample Program which you can use to quickly get AMX up and running in your development environment. The sample can be used as a starting point for the construction of your own AMX application as described in Chapter 3. The AMX Prototyping System (TAPS™) lets you test much of your AMX application on a Windows® platform when your target hardware is not available.

Chapter 4 will introduce you to the *KwikLook*™ Fault Finder, the Windows DLL that adds AMX task-awareness to many of the popular debuggers supported by KADAK. Also described are some of the debugging techniques recommended by KADAK's experienced technical staff for testing real-time software systems.

Chapter 5 offers hints for using the AMX kernel to best advantage and suggests guidelines for proper application design. Chapter 6 discusses some of the C/C++ pitfalls that newcomers to multitasking software often face for the first time.

### Parts List

The AMX Multitasking Executive software is delivered to you in an encrypted form on the AMX CD-ROM which is usually enclosed inside the front cover of the hard copy AMX Reference Manual.

The printed manual consists of five separate guides: this Getting Started guide, the AMX Tool Guide, the AMX Target Guide, the AMX User's Guide and the *KwikLook* User's Guide. The Tool Guide and Target Guide are different for each target processor family. The other guides are common to all 32-bit versions of AMX.

The AMX manuals for all processor families are directly accessible in Adobe Acrobat® PDF format on the AMX CD-ROM. During installation, you will be given the option of installing the manuals for easy online access when using AMX.

The AMX Prototyping System (TAPS) is delivered on the AMX CD-ROM. The TAPS User's Guide in HTML format can be viewed with your document browser. It is directly accessible on the CD-ROM and is always installed with TAPS for online viewing. A printed copy is not provided.

The *KwikLook* Fault Finder is delivered on the AMX CD-ROM. The *KwikLook* User's Guide in Adobe Acrobat PDF format is directly accessible on the CD-ROM and is always installed along with the AMX manuals for online viewing.

The most recent revision of each manual is always available from KADAK's website at [www.kadak.com](http://www.kadak.com).

## 1.2 Installing the AMX Software

### Before You Start

The installation utility will not allow you to install AMX if it detects an existing installation of the same product without first warning you and giving you the option of deleting the previous installation before proceeding. For example, you will get such a warning if you are updating your version of AMX with a newer release.

To preserve an existing installed copy of AMX, rename or move the installation directory of the previous copy of the product before installing the new version.

### Installing AMX

AMX is installed by running the InstallShield® *SETUP.EXE* program located in the root directory of the CD-ROM. From the Windows Start, Run... menu, type *D:\SETUP.EXE* (where *D:* is your CD-ROM drive letter) and press Enter. Alternatively, browse the root directory of the CD-ROM and double click on the *SETUP.EXE* filename or icon.

During installation you will need your AMX CD serial number and product installation key, both of which are recorded on the AMX Parts List packaged with the CD-ROM. They are also recorded on a label attached to the CD-ROM case. The CD serial number is also printed on the face of the CD-ROM.

The **CD serial number** identifies your actual AMX CD-ROM, not the products which it contains. It is the product **installation key** which identifies the specific AMX product which you are entitled to install from the CD-ROM.

The setup utility will lead you through the installation process. You may be requested to identify the software development tools which you plan to use for development. Only the tools specifically supported by KADAK will be listed. You can select one or more of the listed toolsets. If you are not sure which tools you will be using, select them all. After installation, you can always delete support for the unused tools. The supported toolsets are identified in Chapter 1.5.

The installation process will copy the product files into a directory of your choice on the disk drive of your choice. AMX files will be installed in subdirectory *AMXnnn* where the number *nnn* comes from the KADAK part number PNnnn-1 used to identify the AMX product.

Directory	Part	Product
<i>AMX382</i>	PN382-1	AMX PPC32
<i>AMX402</i>	PN402-1	AMX 4-ARM
<i>AMX422</i>	PN422-1	AMX 4-Thumb
<i>AMX442</i>	PN442-1	AMX MA32
<i>AMX512</i>	PN512-1	AMX CFire
<i>AMX532</i>	PN532-1	AMX 68000
<i>AMX722</i>	PN722-1	AMX 386/ET

## Installed AMX Software Components

The AMX installation directory *AMXnnn* will include the following subdirectories:

<i>CFGBLDW</i>	AMX Configuration Builder for Windows
<i>ERR</i>	Error directory used by make process
<i>MAKE</i>	AMX make utilities
<i>MANUALS</i>	AMX documentation
<i>TOOLXX</i>	Toolset dependent files for toolset <i>xx</i>
<i>TOOLXX\CFG</i>	Configuration template files
<i>TOOLXX\DEF</i>	AMX C header and assembler definition files
<i>TOOLXX\LIB</i>	AMX libraries and object files
<i>TOOLXX\SRC</i>	AMX kernel and manager source files
<i>TOOLXX\SAMPLE</i>	AMX Sample Program files

A complete list of the installed AMX files will be found in the product manifest file *MANIFEST.TXT*, a text file present in each *TOOLXX* toolset subdirectory. A separate manifest file is provided for each toolset because there are minor differences in the complement of files required by different tools.

Once AMX has been installed, you will find the following AMX software components in your AMX installation directory.

The **AMX Configuration Builder** is the Windows utility which you will use to specify the AMX features required by your application and the target processor characteristics which AMX must support.

The **AMX Library** is installed ready for use with each of the software development toolsets which KADAK supports.

**Source code** for the AMX Library is also installed so that if, for some reason, you must rebuild the AMX Library, you can do so. Your application will also require access to the AMX header files to compile or assemble modules which reference AMX procedures.

An **AMX Sample Program** is provided as an example of a working AMX application. This sample application is ready for use even if all you have is just a target processor and memory. It will therefore run with little, if any, modification using your debugger's target processor simulator. A variation of the AMX Sample Program is also provided for most of the hardware evaluation boards on which KADAK has tested AMX.

## Installed AMX Manuals

If you install the AMX documentation, this Getting Started manual and the following manuals, all in Adobe Acrobat PDF format, will be installed in your AMX installation subdirectory *AMXnnn\MANUALS*. All of these manuals are then accessible via Windows links in the AMX program group created when AMX is installed. These manuals are also accessible directly on the AMX CD-ROM.

The **AMX Tool Guide** provides guidance for the proper use of AMX with each toolset with which AMX has been tested. The AMX Tool Guide specifies the command line switches which are required when using the command line tools to compile or assemble modules, create libraries and link AMX applications.

The **AMX Target Guide** describes hardware specific requirements and programming considerations that apply to the target processor with which AMX is being used. Processor exceptions, device interrupts, clock drivers and caching issues are all discussed. This guide also illustrates how the AMX Configuration Builder is used to specify the target parameters which adapt AMX to your specific hardware environment. All processor dependent procedures in the AMX Library are documented in this guide.

The **AMX User's Guide** is the chief AMX programming manual. It describes the AMX kernel, its managers and their use. This guide illustrates how the AMX Configuration Builder is used to specify the characteristics of the AMX application you intend to create. It also includes a description of each AMX service procedure, its calling sequence and, where applicable, an illustrative example of proper usage.

The following auxiliary documentation, in Adobe Acrobat PDF format, is also provided with AMX. These manuals are not included in the hard copy AMX Reference Manual.

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 Conversion Guide** describes how to convert AMX applications developed using earlier versions of AMX to operate with the current AMX release. This manual will only be of interest if you are converting from AMX 86 v3.0x, AMX 386 v1.0x or AMX 68000 v2.0x to any of the current 32-bit AMX implementations.

## 1.3 Installing the AMX Prototyping System (TAPS)

### Before You Start

The installation utility will not allow you to install TAPS if it detects an existing TAPS installation without first warning you and giving you the option of deleting the previous installation before proceeding. For example, you will get such a warning if, when updating AMX, you attempt to reinstall TAPS from the CD-ROM containing the newer release of AMX. Do not reinstall TAPS unless specifically directed to do so by the update instructions packaged with the new release of AMX.

To preserve an existing installed copy of TAPS, rename or move the installation directory of the previous copy of TAPS before installing the new version.

### Installing TAPS

TAPS is installed separately from AMX. Follow the same procedure as you used to install AMX. Run the InstallShield *SETUP.EXE* program located in the root directory of the AMX CD-ROM.

Use the same CD serial number as used for AMX but use your TAPS installation key. Both are recorded on the AMX Parts List packaged with the AMX CD-ROM. They are also recorded on a label attached to the CD-ROM case. The CD serial number is also printed on the face of the CD-ROM.

The CD **serial number** identifies your actual AMX CD-ROM, not the products which it contains. It is the TAPS **installation key** which permits you to install TAPS from the CD-ROM.

The setup utility will lead you through the installation process. The installation process will copy the TAPS files into a directory of your choice on the disk drive of your choice. TAPS files will be installed in subdirectory *TAPS302*.

## Installed TAPS Software Components

The TAPS installation subdirectory *TAPS302* will include the following subdirectories:

<i>CFG</i>	AMX Configuration Builder and template file
<i>MANUAL</i>	TAPS online documentation in HTML format
<i>DEF</i>	AMX C header files
<i>LIB</i>	AMX Library
<i>SAMPLE</i>	AMX Sample Program as a Visual Studio project

A complete list of the installed TAPS files will be found in the product manifest file *MANIFEST.TXT*, a text file present in the TAPS installation directory *TAPS302*.

The following TAPS software components will be installed.

The **AMX Configuration Builder** is the Windows utility which you will use to specify the AMX features required by your application. Since TAPS does not require target hardware, there will be no need to specify your target processor characteristics.

**Source code** for the TAPS compatible AMX C header files is provided so that you can use Microsoft<sup>®</sup> Visual C++ to compile your AMX application modules which reference AMX procedures.

The **AMX Library** for TAPS will be used to build an AMX application from within the Microsoft Visual Studio IDE.

The **AMX Sample Program** is provided as a Microsoft Visual Studio project. You can build this sample within the Visual Studio IDE and see first hand how easy it is to create and test a working AMX application using TAPS on a Windows platform.

## Installed TAPS Manual

The TAPS User's Guide in HTML format will be installed in your TAPS installation directory. Use your document browser to view file *TAPS302\MANUAL\TAPS302.HTM*. This manual is also directly accessible on the AMX CD-ROM. A printed copy is not provided.

The TAPS manual is always accessible via a Windows link in the TAPS program group created when TAPS is installed.

## 1.4 Installing the *KwikLook* Fault Finder

### Before You Start

The installation utility will not allow you to install the *KwikLook* Fault Finder if it detects an existing installation without first warning you and giving you the option of deleting the previous installation before proceeding. For example, you will get such a warning if you are updating your version of *KwikLook* with a newer release.

To preserve an existing installed copy of *KwikLook*, rename or move the installation directory of the previous copy of *KwikLook* before installing the new version.

### Installing *KwikLook*

*KwikLook* is installed separately from AMX. Follow the same procedure as you used to install AMX. Run the InstallShield *SETUP.EXE* program located in the root directory of the AMX CD-ROM.

Use the same CD serial number as used for AMX but use your *KwikLook* installation key. Both are recorded on the AMX Parts List packaged with the AMX CD-ROM. They are also recorded on a label attached to the CD-ROM case. The CD serial number is also printed on the face of the CD-ROM.

The CD **serial number** identifies your actual AMX CD-ROM, not the products which it contains. It is the *KwikLook* **installation key** which permits you to install *KwikLook* from the CD-ROM.

The setup utility will lead you through the installation process. The installation process will copy the *KwikLook* files into a directory of your choice on the disk drive of your choice. *KwikLook* files will be installed in subdirectory *KWK302*.

During the installation, you will be asked to identify the debugger with which you plan to use *KwikLook*. Some of the *KwikLook* files may be copied to your debugger's installation directory. In some cases, the duplicates installed for the debugger will be renamed to satisfy the debugger's DLL naming requirements or to permit *KwikLook* to identify the version of AMX which *KwikLook* must support.

#### Toolset Tip

You can install *KwikLook* for more than one task-aware debugger. For example, if you have installed AMX PPC32 to support both MetaWare and Metrowerks tools, then you can install *KwikLook* for both SeeCode and CodeWarrior.

## Installed *KwikLook* Software Components

The *KwikLook* installation subdirectory *KWK302* will include the following subdirectories:

<i>HOST</i>	<i>KwikLook</i> DLL and support files
<i>MANUAL</i>	<i>KwikLook</i> User's Guide in Adobe Acrobat PDF format

A complete list of the installed *KwikLook* files will be found in the product manifest file *MANIFEST.TXT*, a text file present in the *KwikLook* installation directory *KWK302*.

## Installed *KwikLook* Manuals

The *KwikLook* User's Guide in Adobe Acrobat PDF format will be installed in your *KwikLook* installation directory. This manual is also directly accessible on the AMX CD-ROM. A printed copy is included with the AMX Reference Manual.

The *KwikLook* Help Manual will be installed in your *KwikLook* installation directory. This manual can be viewed using the Windows help facility. It is also directly accessible from within *KwikLook* in a context sensitive manner. A printed copy is not provided.

The *KwikLook* manuals are always accessible via Windows links in the *KwikLook* program group created when *KwikLook* is installed.

## Supported Task-Aware Debuggers

<b>AMX Part</b>	<b>Target</b>	<b>Toolset xx</b>	<b>Task-Aware Debugger</b>
PN382-1	PowerPC	<i>DA</i>	Diab-SDS SingleStep
PN382-1	PowerPC	<i>ME</i>	Metrowerks CodeWarrior
PN382-1	PowerPC	<i>MW</i>	MetaWare SeeCode
PN402-1	ARM	<i>ME</i>	Metrowerks CodeWarrior
PN402-1	ARM	<i>MW</i>	MetaWare SeeCode
PN402-1	ARM	<i>RV</i>	ARM Ltd. RealView
PN422-1	Thumb	<i>ME</i>	Metrowerks CodeWarrior
PN422-1	Thumb	<i>MW</i>	MetaWare SeeCode
PN422-1	Thumb	<i>RV</i>	ARM Ltd. RealView
PN442-1	MIPS32	<i>MW</i>	MetaWare SeeCode
PN512-1	ColdFire	<i>DA</i>	Diab-SDS SingleStep
PN512-1	ColdFire	<i>ME</i>	Metrowerks CodeWarrior
PN532-1	68000	<i>DA</i>	Diab-SDS SingleStep
PN532-1	68000	<i>ME</i>	Metrowerks CodeWarrior
PN722-1	80x86	<i>PD</i>	Paradigm Debugger (protected mode)

## 1.5 Choosing Your Toolset

The AMX Multitasking Executive has been built on a PC running Microsoft Windows using the software development tools described in this chapter. Preconstructed AMX Libraries are installed ready for your use with these toolsets. The AMX Tool Guide will identify the most recent release of each toolset tested by KADAK.

To construct your embedded AMX application, you will require a C or C++ compiler, an assembler, a librarian (optional), a linker and/or locator and a remote debugger. The vendors listed below provide these tools.

KADAK assigns a unique two or three character mnemonic *xx* called a **toolset id** to identify each toolset combination with which AMX has been built and tested.

When AMX is installed, the AMX files required for use with toolset *xx* are installed in a toolset subdirectory named *TOOLXX*. Since the toolset ids are unique, there will be multiple *TOOLXX* subdirectories present if you installed support for more than one toolset.

To remove support for toolset *xx*, delete its toolset subdirectory *TOOLXX*. Doing so will have no adverse effect on any of the remaining toolset subdirectories.

Vendor	Processor family	Toolset id	Toolset directory
ARM Ltd. (SDT, ADS)	ARM	<i>RM</i>	<i>AMX402\TOOLRM</i>
	Thumb	<i>RM</i>	<i>AMX422\TOOLRM</i>
ARM Ltd. (RVDS)	ARM	<i>RV</i>	<i>AMX402\TOOLRV</i>
	Thumb	<i>RV</i>	<i>AMX422\TOOLRV</i>
Mentor Graphics	68000	<i>MR</i>	<i>AMX532\TOOLMR</i>
MetaWare	PowerPC	<i>MW</i>	<i>AMX382\TOOLMW</i>
	ARM	<i>MW</i>	<i>AMX402\TOOLMW</i>
	Thumb	<i>MW</i>	<i>AMX422\TOOLMW</i>
	MIPS32	<i>MW</i>	<i>AMX442\TOOLMW</i>
Metrowerks	PowerPC	<i>ME</i>	<i>AMX382\TOOLME</i>
	ARM	<i>ME</i>	<i>AMX402\TOOLME</i>
	Thumb	<i>ME</i>	<i>AMX422\TOOLME</i>
	ColdFire	<i>ME</i>	<i>AMX512\TOOLME</i>
	68000	<i>ME</i>	<i>AMX532\TOOLME</i>
Paradigm Systems (32-bit)	80x86	<i>PD</i>	<i>AMX722\TOOLPD</i>
TASKING (formerly Intermetrics)	68000	<i>IM</i>	<i>AMX532\TOOLIM</i>
Wind River (Diab-SDS tools)	PowerPC	<i>DA</i>	<i>AMX382\TOOLDA</i>
	ColdFire	<i>DA</i>	<i>AMX512\TOOLDA</i>
	68000	<i>DA</i>	<i>AMX532\TOOLDA</i>

## Toolset Caveats

The AMX Tool Guide is meant to serve as a guide to the proper use of the software development tools with which AMX has been used. The guide is NOT meant to replace the manuals provided with the tools. In fact, from time to time, the information in the AMX Tool Guide will be superceded by newer releases of the tools.

KADAK tries to keep the AMX Tool Guide current but the number and frequency of tool revisions makes it very difficult to do so. The following suggestions are offered to allow you to use new tool releases without necessarily waiting for KADAK to validate the tool.

Do not try to mix and match your tools unless they are designed to work together. For example, one vendor's linker cannot necessarily link object modules produced by the another vendor's C compiler, even if the vendors claim to support the same object format.

It is especially important to use tools in proper revision order. For instance, new releases of a linker will usually link previous libraries and object modules. But the old linker may not handle new libraries created with a new copy of the librarian.

If you alter any AMX source module or rebuild any AMX object module using a new release of a tool, it is advisable to rebuild all AMX modules with the new tool. Follow the directions in Appendix D of the AMX User's Guide for rebuilding the AMX Library.

When you make object or library modules, do not expect to generate files which exactly match those delivered by KADAK. Many C compilers, assemblers, linkers and librarians insert source filename and path information in the output modules. They also may insert compilation time and date information in the files. Consequently, two sequential compilations of a single, unaltered file may produce two correct object modules which do not match byte for byte.

You may also find the embedded path information to be very aggravating when you port the libraries to a different machine for testing. You may find that your debugger cannot locate the source code for the module which you are testing because the path used to compile the module does not exist on the test machine.

### Toolset Tip

File extensions *.S*, *.O* and *.A* are used throughout this manual for assembly language, object and library files respectively. Other extensions such as *.ASM*, *.OBJ* or *.LIB* may be used by some toolsets.

## 1.6 Program Groups

When AMX, TAPS or *KwikLook* are installed, a program group is created for quick access via the Windows Start menu to the corresponding installed components. The program group includes the shortcuts described in this chapter.

### Installation Notes

A shortcut called Read Me uses the Windows NotePad utility to open a text file describing the installed product. This is the same text file which is presented for your viewing at the completion of the installation process.

### Uninstall

A shortcut called Uninstall initiates the removal of the installed files.

### AMX Configuration Manager

A shortcut called AMX Configuration Manager is created in the AMX program group to start the AMX Configuration Manager to create or edit your AMX parameter files.

### Manuals

The AMX and *KwikLook* manuals are provided in Adobe Acrobat PDF format. The installer will create a shortcut for each manual to invoke the Acrobat Reader for online viewing of the manual. A copy of the Acrobat Reader is provided with AMX in case you need it installed.

A shortcut called KwikLook Help will be created so that you can view the *KwikLook* Help Manual using the Windows help facility.

A shortcut called TAPS User's Guide will be created so that you can view the TAPS manual using your HTML document browser.

If support for your toolset's IDE is provided with AMX, a shortcut called IDE Guide will be created so that you can use your HTML document browser to view the instructions for using AMX within that IDE.

### Task-Aware Debuggers

When you install *KwikLook* for a task-aware debugger, the program group will usually contain a shortcut to start the debugger for task-aware operation with *KwikLook*. In some cases, additional shortcuts will be created to launch different variants of the debugger for use with a simulator or remote target monitor or with a special BDM or JTAG connection.

## 1.7 Uninstalling AMX, TAPS or *KwikLook*

When AMX, TAPS or *KwikLook* are installed, a program group is created for quick access to their components via the Windows Start menu. A shortcut called Uninstall is created in each program group to initiate the removal of the installed files.

You can also uninstall AMX, TAPS or *KwikLook* using the Add/Remove Programs service available in the Windows Control Panel.

### AMX

If you uninstall the version of AMX identified by KADAK part number PNnnn-1, the *AMXnnn* installation subdirectory will be deleted, including all files contained therein.

### TAPS

If you uninstall TAPS, the *TAPS302* installation subdirectory will be deleted, including all files contained therein.

### *KwikLook*

If you uninstall *KwikLook*, the *KWK302* installation subdirectory will be deleted, including all files contained therein.

The *KwikLook* files, if any, which are duplicated in the installation directory of your task-aware debugger are not removed. To delete these files so that your debugger installation has no *KwikLook* remnants, follow the instructions provided in the *KwikLook* User's Guide.

#### **Warning!!!**

Do not uninstall AMX, TAPS or *KwikLook* without first making a backup copy of the corresponding installation directory. This is especially important if you have made alterations to any AMX, TAPS or *KwikLook* files or if you have stored files of your own in any of their installation subdirectories.

## 2. AMX Sample Program

### 2.1 Sample Program Operation

#### System Description

An AMX Sample Program is provided to illustrate the ease with which an AMX system can be created. The sample includes all of the software elements which make up a real AMX system. A listing of the actual sample code is provided in this chapter.

In order to let you get started quickly, the AMX Sample Program is designed to operate with only a processor and memory. Timing can be "simulated" so that there is not even a need for a hardware clock. Although a console device is required for displaying text strings one line at a time, even it can be omitted if necessary.

The AMX Sample Program uses two AMX timers to periodically send messages at different priorities to a message exchange. A printing task extracts messages from the exchange, decodes each message and displays the information as a text string on the console device. A low priority task delays for sixty seconds and then initiates a shutdown to terminate execution of the AMX Sample Program. A very low priority background task simulates AMX clock ticks if a hardware clock is not available.

#### Components

The system is implemented using the following AMX elements: two Restart Procedures, one Exit Procedure, a Print Task, a Shutdown Task, a Background Task, two timers and one message exchange.

The application Restart Procedures, Exit Procedure, Timer Procedures and tasks are coded in C. Because there is so little code required to implement this system, it is lumped into the single source file *CJSAMPLE.C*.

Every AMX application includes two configuration modules. One describes the system from the AMX perspective. The other identifies the target hardware environment.

The AMX System Configuration Module (a C source file) is constructed from the parameters in a User Parameter File, a text file which can be created and edited using the AMX Configuration Builder. In this sample, the Restart and Exit Procedures, the Print Task, the timers and the message exchange are predefined in this module.

The AMX Target Configuration Module (an assembly language module) is constructed from the parameters in a Target Parameter File, a text file which can also be created and edited using the AMX Configuration Builder. In this sample, the simulated clock ISP is identified in the Target Parameter File.

These three components must be compiled and assembled and linked with the AMX Library to form an executable program which can be run by your target hardware simulator or downloaded to your target hardware for execution.

## Operation

The AMX Sample Program begins like a regular C program in the `main()` function. The AMX RTOS is launched from `main()`. After AMX has initialized itself and built all of the predefined tasks and timers and the message exchange, it calls each of the application's two Restart Procedures.

The first Restart Procedure starts the two timers, sends a sign-on message to the message exchange and triggers the Print Task.

The second Restart Procedure creates the Shutdown Task and the Background Task and triggers both of them.

Once the Restart Procedures have been executed, AMX enters its task scheduler and thereafter assumes full responsibility for proper prioritized, multitasking execution of the application tasks.

The Print Task waits for messages to arrive on the message exchange. As each message is received, the task translates the information in the message into a text string and displays the string on the console device.

As soon as the Print Task finishes displaying the sign-on message, it blocks waiting for the next message. The Shutdown Task begins and delays for one minute before it initiates a shutdown of the AMX system.

When the Print Task and Shutdown Task are no longer active, the Background Task runs. If the sample has been built to use a real hardware clock, the task simply ends. Otherwise the task runs forever, simulating hardware clock ticks.

The two timers are periodic. As each timer expires, AMX restarts the timer and calls the associated application Timer Procedure. Each of these sends a message to the message exchange. The message identifies the timer (timer 1 or timer 2) and provides the value of the AMX clock tick counter at the instant the timer expired.

When the Shutdown Task initiates the system shutdown, AMX calls the application Exit Procedure which stops the two timers, sends a sign-off message to the message exchange and then waits for the Print Task to print the sign-off message.

Once the Exit Procedure completes, AMX shuts down and returns to the `main()` function from which it was launched.

## Console Device I/O

The AMX Sample Program requires a console output device to print simple text messages, one line at a time. All console output is directed through a simple console interface which supports only single character output. One of three device I/O methods can be used.

The Sample Program can be built to use the C library function `putchar()` for character output. This method allows the Sample Program to be run by debuggers and/or target hardware simulators which provide a console window accessible via standard C runtime library functions.

For systems without any console support, the Sample Program can be built to insert all console characters into a character array `cj_records[]`. Pointers to the text strings are inserted into a record array `cj_recordlist[]`. You can then use your debugger to breakpoint in the `main()` function upon return from AMX and display the content of either of these arrays.

If you are using any of the evaluation boards on which AMX has been exercised by KADAK, you can build the Sample Program to use the board support module and serial I/O (UART) driver delivered with AMX. In this case, console I/O will be directed through the serial driver's polled I/O interface function `chuart()` to the console connected to one of the serial ports on the evaluation board.

## Sample Program Output

The following messages appearing on the console terminal in the sequence shown confirm the proper operation of the AMX Sample Program.

```
AMX Sample System begins.  
  
Timer 2: message priority 2 at 1 ticks  
Timer 1: message priority 3 at 1 ticks  
Timer 1: message priority 3 at 101 ticks  
Timer 2: message priority 2 at 201 ticks  
Timer 1: message priority 3 at 201 ticks  
Timer 1: message priority 3 at 301 ticks  
Timer 2: message priority 2 at 401 ticks  
Timer 1: message priority 3 at 401 ticks  
.  
.  
.  
Timer 2: message priority 2 at 5601 ticks  
Timer 1: message priority 3 at 5601 ticks  
Timer 1: message priority 3 at 5701 ticks  
Timer 2: message priority 2 at 5801 ticks  
Timer 1: message priority 3 at 5801 ticks  
Timer 1: message priority 3 at 5901 ticks  
  
AMX Sample System ends.
```

## Sample Program Listing

The application code for the AMX Sample Program can be viewed in source file *CJSAMPLE.C*. The following partial listing of that file illustrates the main elements which make up this simple AMX system. For simplicity, details of the console I/O interface and the clock simulation by the Background Task have been omitted from the listing.

```
#include <stdio.h>                /* Standard I/O Header      */
#include "cjzzz.h"                /* AMX Headers              */

/* External references            */
extern CJ_ID prtskid;            /* Print Task id            */
extern CJ_ID prtmxid;           /* Message exchange id     */
extern CJ_ID tmr1id;            /* Timer 1 id               */
extern CJ_ID tmr2id;           /* Timer 2 id               */
void CJ_CCPP chbrdinit(void);    /* Initialize board         */

/* Forward references            */
void CJ_CCPP sdtask(void);       /* Shutdown Task            */
void CJ_CCPP bgtask(void);      /* Background Task          */
void msgsend(char *msgp, int priority, int wait); /* Send a message          */
void concfg(void);              /* Configure serial I/O device */
void conouts(char *bufferp);    /* Output character string  */

/* Local variables              */
#define SDPRIORITY 30           /* Shutdown Task priority  */
#define SDSTKSZ 2048           /* Shutdown Task stack     */
static unsigned int sdstack[SDSTKSZ/sizeof(int)];

#define BGPRIORITY 40           /* Background Task priority */
#define BGSTKSZ 2048           /* Background Task stack   */
static unsigned int bgstack[BGSTKSZ/sizeof(int)];

/* Application message structure */
struct appmsg {
    char *string;               /* Message string          */
    CJ_T32U tick;               /* AMX tick value          */
};

/* Main Program                 */

int main()
{
    chbrdinit();                /* Initialize board (if needed)*/
    cjkslaunch();               /* Launch AMX               */
    return (0);
}
```



```

/* Print Task */
void CJ_CCPP prtask(void)
{
    char    buffer[80];
    union {
        struct cjxmsg maxmsg;    /* Maximum sized AMX message */
        struct appmsg amxmsg;    /* Application message */
    } umsg;

    concfg();                    /* Configure I/O device */

    for (;;) {                  /* Wait forever for messages */
        cjmxwait(prtmxid, &umsg, 0, 0);

                                /* Format string into buffer */
        sprintf(buffer, umsg.amxmsg.string, umsg.amxmsg.tick);

        conouts(buffer);        /* Print the string */

        cjtmsgack(CJ_EROK);     /* Acknowledge message */
    }
}

/* Shutdown Task */
void CJ_CCPP sdtask(void)
{
    cjtmdelay(cjtmconvert(60000L)); /* Wait one minute */
    cjksleave(0, CJ_NULL);         /* Shut down AMX */
}

/* Exit Procedure */
/* NOTE: Since cjksleave is called by the Shutdown Task,
/* this procedure executes in the context of that task.
void CJ_CCPP exproc(void)
{
    cjtmmwrite(tmrlid, 0L);        /* Stop timers */

                                /* Wait for sign-off message */
    msgsend("\nAMX Sample System ends.\n", 0, CJ_YES);
}

```



## 2.2 Building the AMX Sample Program

### Quick Start

Before you can build the AMX Sample Program, you must first be familiar with your software development tools. You must know how to run the C/C++ compiler, the assembler and the linker/locator to construct an executable program module. You must also know how to use the debugger to load and execute the program, whether using a target simulator or real hardware.

The AMX Sample Program is provided in several forms ready for use with toolset *xx*. Each variant supports a particular hardware environment in which the sample has been tested. The source code for each implementation resides in its own subdirectory within toolset directory *TOOLXX\SAMPLE*.

The source code for the simplest AMX Sample Program will be found in toolset directory *TOOLXX\SAMPLE\MEFIRST*. As the directory name suggests, you should start with this example because it has no hardware dependencies beyond the need for a simple console.

There are four steps needed to build the AMX Sample Program:

- Compile the AMX Sample Program                   file *CJSAMPLE.C*
- Compile the AMX System Configuration Module   file *CJSAMSCF.C*
- Assemble the AMX Target Configuration Module   file *CJSAMTCF.S*
- Link and locate the AMX Sample Program to create an executable load module

You can build the AMX Sample Program using command line tools within a Windows Command Prompt window. At the command line prompt, go to toolset directory *TOOLXX\SAMPLE\MEFIRST* and run your make utility to build the program from make specification file *CJSAMPLE.MAK*. The make utility will create a load module (*CJSAMPLE.OUT* or equivalent) suitable for execution under the control of the debugger provided with toolset *xx*.

## Using Your Tools

If you are using toolset *xx*, you should examine all of the files in toolset directory *TOOLXX\SAMPLE\MEFIRST*. In particular, review the toolset dependent **tailoring file** named *CJZZZCC.INC*. This file, included within make file *CJSAMPLE.MAK*, is used by the make utility to run the compiler, assembler and linker for toolset *xx*.

Review tailoring file *CJZZZCC.INC* to see exactly how the compiler and assembler are used to create the object files which form the sample system. Observe the recommended command line switches for compiling and assembling application modules for use with AMX using the most recent tools for toolset *xx*. Also note the command line switches required to link and locate the final load module. These switches are documented in the AMX Tool Guide.

Within directory *TOOLXX\SAMPLE\MEFIRST* you will also observe one or more toolset dependent link, locate, memory or command specification files. These files have the filename *CJSAMPLE* and an extension such as *LKS*, *LOC*, *LCF*, *CFG* or *CMD*. The files specify the target memory layout and indicate the order in which object and library files are to be linked.

The link/locate files are used by the make file *CJSAMPLE.MAK* to create the AMX Sample Program load module. You should review these files to observe the special directives, if any, required by the linker and/or locator provided with toolset *xx*.

## Memory Layout

The AMX Sample Program from directory *TOOLXX\SAMPLE\MEFIRST* can be used without modification if toolset *xx* provides a Windows based target processor simulator. Otherwise, the sample must be downloaded and executed on a hardware platform.

Examine source file *CJSAMPLE.C* and look for the comment block which indicates whether the sample can be run using the toolset's simulator or must be downloaded to target hardware. In the latter case, the sample will be linked using the memory configuration common to the greatest number of boards available at KADAK. You may have to revise the sample link/locate specification file to adapt the memory layout for your specific board.

## Console Output

If the sample can be run using a simulator and the simulator supports standard C console output, then the sample program will direct its output to the simulator's console window. Otherwise, the sample program will direct its console output to the character array *cj\_records[]* in target memory.

Examine source file *CJSAMPLE.C* and look for the definition of symbol *K\_CONSOLE*. If it specifies *K\_STDIO*, then the sample will use the standard C library function *putchar()* for console output. If it specifies *K\_RECORD*, then console output will be inserted into character array *cj\_records[]*.

## Running the Sample Program

Use the debugger provided with toolset *xx* to load the AMX Sample Program into the processor's target memory, whether simulated or on a real hardware platform. Start the sample with a breakpoint on the entry to function *main()* to confirm that you have successfully executed the C startup code and are ready to start AMX. Then proceed with a breakpoint following the call to procedure *cjkslaunch* which launches AMX.

If the debugger or simulator console window is being used for console output, you will see the AMX Sample Program messages appearing in that window at periodic intervals. If the console window output is buffered, the display of messages may occur in bursts.

If console output is being directed to character array *cj\_records[]* in target memory, there will be no visible evidence that the program is running.

The AMX Sample Program will continue to execute until the AMX clock indicates that one minute has elapsed. Since the AMX clock ticks are generated by a Background Task and not by a real hardware clock, the AMX clock will not keep proper time. When running on an actual high frequency processor, the AMX clock will appear to run quickly. When running under a simulator which mimics a low frequency processor, the AMX clock will appear to run slowly.

When the AMX system shuts down, your final breakpoint in *main()* will be reached. If there has been no visible console output, you can use the debugger to dump the text in character array *cj\_records[]* or the strings in pointer array *cj\_recordlist[]* to confirm proper operation of the sample.

## Board Specific Sample Programs

The AMX Sample Program is also delivered ready for use on each of the hardware platforms on which AMX has been tested by KADAK. Each of the board specific examples is located in a separate subdirectory within the toolset dependent directory *TOOLXX\SAMPLE*. All of the files needed to build the AMX Sample Program for a specific board are located in the subdirectory devoted to that board. The boards and their subdirectories are listed in the AMX product manifest file *TOOLXX\MANIFEST.TXT*.

A number of additional source files are required to adapt the AMX Sample Program for use on a particular board. These target sensitive files include a clock driver, a serial driver and a board support module.

The **AMX Clock Driver** uses a hardware timer embedded in the processor chip, or interfaced to it, to provide the fundamental timing source for AMX. These drivers are described in Chapter 5 of the AMX Target Guide. The clock driver is a separate C source file which must be compiled and linked as part of the Sample Program.

An **AMX serial driver** is provided for the UART device, if any, embedded in the processor chip or interfaced to it. The driver operates in a polled fashion to transmit characters to or receive characters from an attached console terminal. The serial driver is a separate C source file which must be compiled and linked as part of the Sample Program. By default, the driver sets the serial port to operate at 9600 baud with one start bit, one stop bit and 8-bit characters without parity. To alter the configuration of the serial device, you must edit the driver source file following the instructions provided within the file. All serial I/O operations offered by the driver are accessed through the driver's chip support function *chuart()*.

An **AMX board support module** includes the minimal services needed to use AMX on a particular hardware platform. Most of these modules include a chip support function *chbrdinit()* which must be called from *main()* before AMX is launched. The interrupt controller, if present, is conditioned to inhibit all interrupts except the hardware timer interrupt being used for the AMX clock. The function also completes any special board setup required by AMX but not done by an on-board monitor or by the C startup code. The board support module is a C or assembly language source file which must be compiled or assembled and linked as part of the Sample Program.

You can build the board specific AMX Sample Program using command line tools within a Windows Command Prompt window. At the command line prompt, go to the board specific subdirectory in toolset directory *TOOLXX\SAMPLE* and run your make utility to build the program from make specification file *CJSAMPLE.MAK*. The make utility will create a load module (*CJSAMPLE.OUT* or equivalent) suitable for execution under the control of the debugger provided with toolset *XX*.

You should examine tailoring file *CJZZZCC.INC* to identify the command line switches which are used to compile or assemble each of the board specific source files and to link the final load module. You should also examine the link/locate specification files to determine the recommended order in which the object files should be linked.

## 2.3 Using a Toolset IDE

With proper care, you can build the AMX Sample Program from within the Integrated Development Environment (IDE) provided by toolset *xx*. To do so, you must create a project and import the relevant source files according to the rules established by the IDE. Import AMX Sample Program source file *CJSAMPLE.C*, the System Configuration Module *CJSAMSCF.C* and board specific C files into the list of C/C++ source modules. Import the Target Configuration Module *CJSAMTCF.S* and board specific assembly language files into the list of assembly language source modules.

To compile or assemble the source files, the project must have access to the AMX header files. It may be sufficient to specify the path to directory *TOOLXX\DEF* for include file searches. Alternatively, you may have to import the AMX header files from directory *TOOLXX\DEF* into a list of include files.

To compile or assemble the source files, you must set the appropriate command options. The required command line switches are documented in the AMX Tool Guide and illustrated in tailoring file *CJZZZCC.INC*. The equivalent IDE switches may have to be established within the IDE using check boxes, radio buttons, pull down lists or custom command line dialogs.

To produce an executable module, the project must have access to the object files to be linked, to the AMX Library in directory *TOOLXX\LIB* and to the relevant C/C++ runtime libraries. In some cases, you may also have to link toolset dependent AMX object files from directory *TOOLXX\LIB* and/or special C/C++ library object modules. It may be sufficient to provide the path to these libraries and object files. Alternatively, you may have to import the libraries and/or library object files into a list of required modules.

You may be able to adapt the link/locate specification file provided with the AMX Sample Program to specify the memory layout for the executable module. If not, you will have to define the hardware memory layout in a manner dictated by the IDE.

Once the project is specified, the build is usually initiated with a simple press of a toolbar button. However, there may still be a number of IDE or debugger configuration settings which must be established before the debugger can successfully load and execute the resulting AMX Sample Program load module.

### Toolset Tip

Browse the AMX installation directory for a subdirectory named *MANUALS\TOOLXX\IDE*. If the directory exists, it will contain the file *AMX\_XX.HTM*, a document in HTML format describing in complete detail how to build an AMX application using the IDE for toolset *xx*.

## 2.4 The TAPS Sample Program

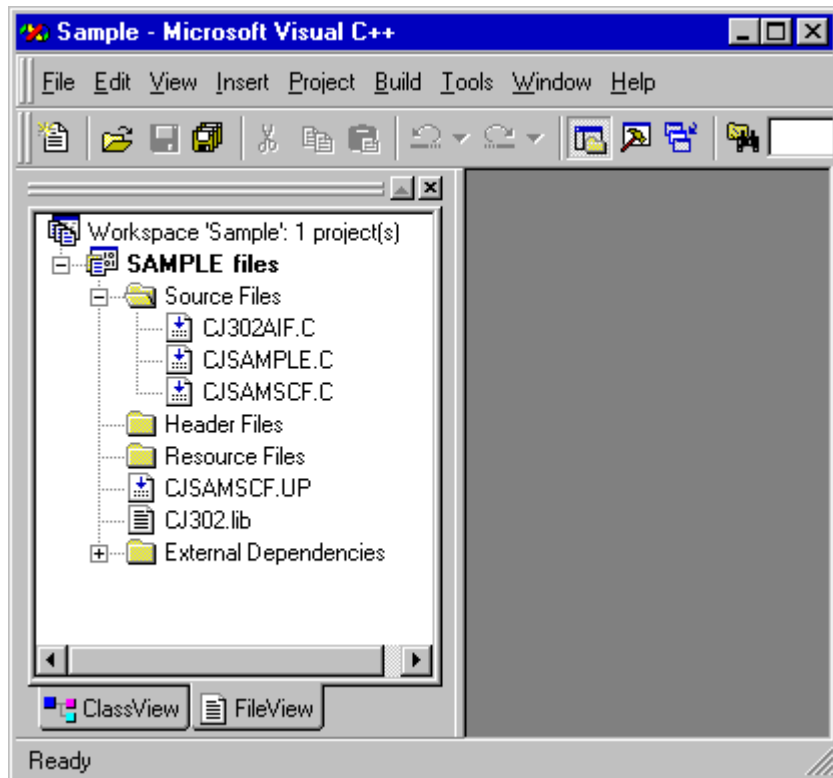
The AMX Prototyping System (TAPS) includes a copy of the AMX Sample Program ready for use on a Windows workstation using the Microsoft Visual Studio. The TAPS installation includes a Visual Studio project with all of the components needed to create and run the AMX sample.

From the Visual Studio File menu, select Open... and browse to find the TAPS sample project file `TAPS302\SAMPLE\SAMPLE.DSP`. The project file list will include the required source files, organized as illustrated in the screen shot shown below.

To build and execute the AMX Sample Program, simply click on the Go button on the Visual Studio toolbar. All source files will be compiled and linked with the TAPS version of the AMX Library. The resulting AMX Sample Program will then be loaded and executed by the Visual Debugger.

The TAPS version of the AMX Sample Program uses the simulated clock provided by TAPS. Character output is directed to the TAPS console window.

The AMX Sample Program is constructed just like any other AMX application intended for use under TAPS. Step by step instructions are presented in the TAPS User's Guide. You can access the manual directly from the Windows Start menu via the link in the TAPS program group. Alternatively, you can view the manual by using your Windows browser to open the installed TAPS file `TAPS302\MANUAL\TAPS302.HTM`.



This page left blank intentionally.

## 3. Creating an AMX Application

### 3.1 The AMX Configuration Process

Before you can construct any AMX application, you must first decide how to best use AMX for your intended purpose. What tasks will you require and how will they interact? Will your tasks need timers, semaphores or mailboxes or any of the many other services offered by AMX? These are but a few of the questions which your design must address.

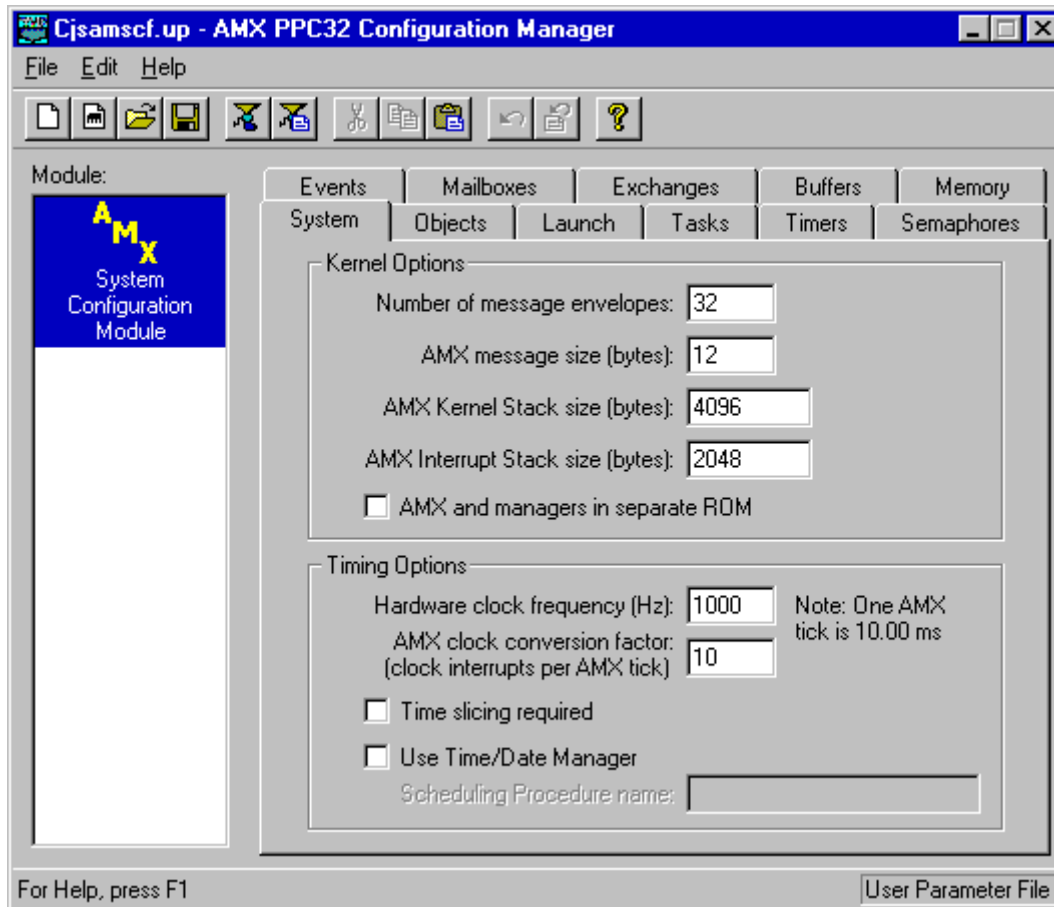
It is assumed that you will read the AMX User's Guide to become familiar with the AMX kernel, its managers and the subset of features best suited for your application. You will also have to become familiar with the material in the AMX Target Guide so that you can incorporate the necessary AMX support for your target hardware configuration.

Then, with your software design in hand, you can use the AMX Configuration Manager to create the configuration files on which your AMX application depends.

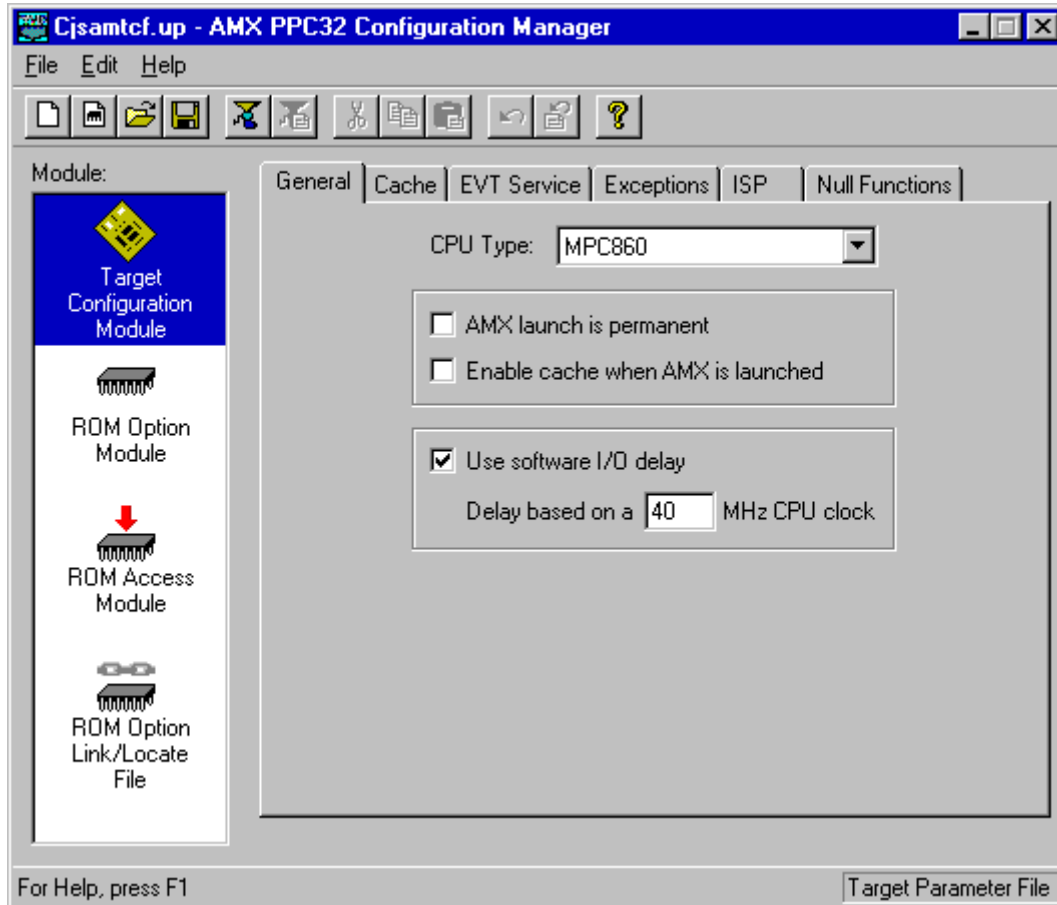
#### The AMX Configuration Manager

If you created an AMX program folder when you installed AMX, you can start the AMX Configuration Manager from the Windows Start menu. Locate the AMX program folder and select the AMX Configuration Manager from the menu. Alternatively, select Run... from the Windows Start menu, browse to the AMX installation directory *AMXnnn\CFGBLDW* and run the program *CJnnnCM.EXE*.

The Configuration Manager is used to construct your AMX System Configuration Module from a User Parameter File as described in Chapter 15 of the AMX User's Guide. To create a new configuration, select New User Parameter File from the File menu. To edit an existing configuration such as that used for the AMX Sample Program, select Open... from the File menu and browse for file *CJSAMSCF.UP* in one of the board dependent directories in toolset directory *TOOLXX\SAMPLE*.



The Configuration Manager is also used to construct your AMX Target Configuration Module from a Target Parameter File as described in Chapter 16 of the AMX User's Guide. To create a new configuration, select New Target Parameter File from the File menu. To edit an existing configuration such as that used for the AMX Sample Program, select Open... from the File menu and browse for file *CJSAMTCF.UP* in one of the board dependent directories in toolset directory *TOOLXX\SAMPLE*.



## Finding Template Files

The AMX Configuration Manager generates the System Configuration Module from a C language template file provided with AMX. It generates the Target Configuration Module from an assembly language template file. Because the template source files are toolset dependent, the files for toolset *XX* are installed in toolset directory *TOOLXX\CFG*.

The AMX Configuration Manager maintains a Windows registry entry which identifies the filename and location of the template file for each module which the manager can generate. If the manager determines that the registry entry does not exist, it creates the entry and installs defaults for all of its template files. The defaults are derived as follows.

If you have only installed AMX support for toolset *XX*, then the manager will determine that the default template files reside in toolset directory *TOOLXX\CFG*.

However, if you have installed support for multiple toolsets, you will have multiple *TOOLXX* directories, each with its own unique toolset id *XX*. In this case, the manager will pick one of the toolsets and choose the template files from the *CFG* directory for that toolset as its defaults.

To examine or revise any of the manager's template file choices, select Templates... from the File menu. To view the template used to generate a particular file, pick the selector icon for that file from the module list. You can then edit the template's path and filename or browse to find the equivalent template file for a different toolset. If you need assistance, press the manager's F1 help key while viewing its Templates dialog.

## The UP Parameter File Extension

KADAK uses the file extension *UP* to identify text files containing user parameters. Whenever the AMX Configuration Manager runs, it registers file extension *UP* with Windows so that the manager can be invoked to edit a parameter file by simply double clicking on the parameter file's name.

If you use more than one version of AMX, each with its own Configuration Manager, you should not double click on parameter filenames to edit the parameter files. Windows will simply run the most recently used manager, whether or not it is the proper manager to be used to edit that particular parameter file. This situation would arise if you were concurrently developing AMX applications for different target processors. For example, you might be using AMX PPC32 in a PowerPC product and AMX 386/ET on a PC. The situation would also exist if you are using two different releases of the same AMX kernel to maintain different revisions of your AMX based product.

If you must use multiple configuration managers, always start the manager of interest first and then use its File menu or toolbar icon to open the parameter file to be edited. Alternatively, you can drag the parameter file directly into the manager of interest, if it is running, or onto the manager's program icon, link or filename.

## 3.2 Building Your AMX Application

The sheer volume of detail provided with AMX may at first be daunting. However, constructing an AMX application is actually a fairly simple process, once you have decided how to best use AMX for your intended purpose.

It is recommended that you adapt elements of the AMX Sample Program to operate on your target hardware, thereby using the sample as a template for the creation of your own AMX application. To assist you in this process, the following guidelines are offered for using toolset *xx* to develop your AMX system.

1. Review the board specific implementations of the AMX Sample Program in the board subdirectories of toolset directory *TOOLXX\SAMPLE*. Find the board with the processor variant and/or hardware complement which most closely matches yours. Copy the files from that board directory into your own working directory.
2. Most AMX applications will use a hardware clock for timing. Review the AMX Clock Driver used by the sample that you selected in Step 1. If it matches your hardware clock, keep it. Otherwise, see Chapter 5 of the AMX Target Guide for a complete description of each of the available clock drivers. Select the driver which most closely matches your target hardware clock.

Each AMX Clock Driver is provided as a C source file. The clock driver may depend on low level services provided by a board support module. The source file for the clock driver and its board support module (if any) will be found in one of the board specific subdirectories in toolset directory *TOOLXX\SAMPLE*. Copy these files to your working directory.

If necessary, edit the selected clock driver and its board support module to match your hardware specifications. In particular, be sure to configure your hardware clock to generate interrupts at the frequency required by your application.

Your AMX Target Configuration Module (see Step 6) will include a clock Interrupt Service Procedure (ISP) which dismisses the clock interrupt, keeps the hardware clock running and informs AMX that a hardware clock tick has occurred. The AMX Configuration Manager automatically generates this clock ISP for you.

Some hardware clocks are so simple that the code to dismiss the clock interrupt can be generated directly into the AMX Target Configuration Module by the AMX Configuration Manager. In other cases, a clock interrupt handler in the clock driver source module will be called upon to dismiss the clock interrupt and keep the hardware clock running. In either case, the clock ISP must be declared as specified in the clock driver description in Chapter 5 of the AMX Target Guide.

3. The AMX Sample Program includes a board support module which provides the minimal board support services required to use AMX and its clock driver on a particular hardware platform. If you already have a board support module from Step 2, skip to Step 4. If not, review the one for the sample selected in Step 1. If it is suitable for use with your hardware, keep it. Otherwise, replace it with the alternate board support module (if any) which provides a better match. If necessary, edit the selected board support module to adapt it for use with your hardware.

4. If you have a serial port available in your hardware configuration, you may wish to use it for simple console I/O operations while testing your AMX application. One of the AMX serial drivers may be suitable for this purpose. Examine the serial drivers in the board directories and copy the one for the UART most similar to yours. If necessary, edit the serial driver to match your device requirements.

Chip support function `chuart()` is used to access the services provided by the serial driver. Your application must call this function to initialize the serial interface. Thereafter, the function can be used to sense the status of the device, transmit characters and fetch received characters.

5. Use the AMX Configuration Manager (see Chapter 3.1) to create a User Parameter File describing your AMX application requirements. Then, from the File menu, select Generate... to create your System Configuration Module, a C source file that must be compiled and linked with your application. This process is described in greater detail in Chapter 15 of the AMX User's Guide.

A good starting point is to use the Configuration Manager to edit the User Parameter File `CJSAMSCF.UP` provided with the sample program that you selected in Step 1. You can view the system parameters that were used to build the sample and, if appropriate, simply edit the settings to describe your own AMX application.

6. Use the AMX Configuration Manager (see Chapter 3.1) to create a Target Parameter File defining your target hardware, clock ISP and application ISPs. From the File menu, select Generate... to create your Target Configuration Module, an assembly language source file that must be assembled and linked with your application. This process is introduced in Chapter 16 of the AMX User's Guide. The processor dependent details are described in Chapter 4 of the AMX Target Guide.

A good starting point is to use the Configuration Manager to edit the Target Parameter File `CJSAMTCF.UP` provided with the sample program that you selected in Step 1. You can view the target parameters that were used to build the sample and, if appropriate, simply edit the settings to describe your hardware configuration.

Pay particular attention to the manner in which the AMX clock is specified in the ISP Definition window. Your clock ISP definition must meet the specifications for the AMX Clock Driver that you selected in Step 2.

7. Your AMX application must have a source module which includes the `main()` function from which AMX is eventually launched. The AMX System Configuration Module created in Step 5 must specify at least one Restart Procedure which initiates some operation (such as triggering a task) to start your system. It is recommended that you use the AMX Sample Program source file `CJSAMPLE.C` as a template for your main application module, stripping the variable and function declarations for which you have no need.

Review the AMX startup module `TOOLXX\SRC\CJnnnUF.C` and, if your needs warrant, add enhancements to the default AMX error handling procedures.

8. You must compile or assemble all of the source modules which make up your AMX application. Besides your own application modules, these include:

AMX Clock Driver	Step 2
AMX board support module	Step 3
AMX serial driver (optional)	Step 4
AMX System Configuration Module	Step 5
AMX Target Configuration Module	Step 6
AMX startup module <i>CJnnnUF.C</i>	Step 7
Main application module	Step 7

Link the resulting object modules with the AMX Library and your C Library to create your AMX application load module.

The AMX Tool Guide describes how to use the assembler, compiler and linker for toolset *xx* when building an AMX application.

### The Build Process

The AMX Sample Program can be constructed by a make utility from the make specification file *CJSAMPLE.MAK*. Recommendations for building the sample within the Integrated Development Environment (IDE) for a particular toolset were also presented in Chapter 2.3. Either of these methods can be used to construct your own AMX application. The choice is yours.

You should examine tailoring file *CJZZZCC.INC* to identify the command line switches which are used to compile or assemble the source files and to link the final load module. You should also examine the toolset dependent link/locate specification files to determine the recommended order in which the object files should be linked.

It is recommended that you construct your AMX application in your own directory, separate from the AMX installation directory. You can then set Windows environment variables to provide the path information needed by your software development tools to find the AMX files (header files, object modules and libraries) required to build your system. If necessary, add the same AMX path information to your make file or to your IDE configuration settings.

The AMX System Configuration Module and Target Configuration Module are source files which must be generated using the AMX Configuration Manager. The manager is an interactive tool which you must use outside your build process, similar to the way you use a text editor to edit other source files. The configuration files generated by the manager should be kept with your other application source files so that each can be compiled or assembled as just another component of your AMX system.

#### Toolset Tip

When building your AMX application, be sure to follow the toolset specific guidelines presented in the AMX Tool Guide.

### 3.3 A Make File for AMX Applications

If you use a *MAKE* utility to control the construction of your AMX application, you should adhere to the following guidelines.

The following discussions assume that you have installed the AMX product with part number PNnnn-1 in directory *C:\KADAK\AMXnnn* as described in Chapter 1.2. Your make specification file should define macros such as *OSPATH* to provide the complete path to the AMX installation directory and *YOURSRC* for access to your source files.

```
OSPATH = C:\KADAK\AMXnnn
YOURSRC = C:\YOURAPP\SOURCE
```

#### AMX Header Dependencies

Do not blindly list the generic AMX include file *CJZZZ.H* as the only AMX dependency in your make specification. Your C files include *CJZZZ.H*, a duplicate of file *CJnnn.H*, for convenience and to ease the porting of your application to other versions of AMX.

However, the file *CJZZZ.H* is the file least subject to change. It is the AMX header files with names like *CJnnnXXX.H* which file *CJZZZ.H* includes that are most likely to undergo future modification. Therefore, examine header file *CJZZZ.H* and identify the AMX header files which it unconditionally includes. Declare your application files to be dependent on this subset of the AMX header files.

Your application files will not be dependent on the AMX header files which are conditionally included by file *CJZZZ.H*. Those files are only used when compiling AMX source modules. For this reason, your AMX System Configuration Module should be declared dependent on all of the files listed in the generic file *CJZZZ.H*.

In your dependency lists, be sure to provide the path information required by the make utility to locate the AMX header files for toolset *xx* in AMX installation directory *\$(OSPATH)\TOOLXX\DEF*.

#### System Configuration Module

Your AMX System Configuration Module *SYSCFG.C* can be created using the AMX Configuration Manager. However, you can alternatively generate and compile the module under the control of your *MAKE* utility as follows.

Use the AMX Configuration Manager to create or edit your AMX User Parameter File *SYSCFG.UP*. Do not ask the Configuration Manager to generate the source module *SYSCFG.C*. That will be done in your make specification file.

Declare your System Configuration Module *SYSCFG.O* to depend on the User Parameter File *SYSCFG.UP* and on the complete list of AMX header files listed in the generic file *CJZZZ.H*. The make directive to create *SYSCFG.O* consists of two statements, one to make the source file *SYSCFG.C* and one to compile it.

The System Configuration Module source file *SYSCFG.C* for use with toolset *XX* can be generated using the AMX Configuration Generator as follows:

```
$(OSPATH)\CFGBLDW\CJnnnCG.EXE $(YOURSRC)\SYSCFG.UP  
$(OSPATH)\TOOLXX\CFG\CJnnnCG.CT $(YOURSRC)\SYSCFG.C
```

The Configuration Generator *CJnnnCG.EXE* combines the information from your User Parameter File *SYSCFG.UP* with the AMX System Configuration Template File *CJnnnCG.CT* to produce file *SYSCFG.C*.

Your make specification file must also include the directive to compile source module *SYSCFG.C* as described in the AMX Tool Guide for toolset *XX*.

## Target Configuration Module

Your AMX Target Configuration Module *HDWCFG.S* can be created using the AMX Configuration Manager. However, you can alternatively generate and compile the module under the control of your *MAKE* utility as follows.

Use the AMX Configuration Manager to create or edit your AMX Target Parameter File *HDWCFG.UP*. Do not ask the Configuration Manager to generate the source module *HDWCFG.S*. That will be done in your make specification file.

Declare your Target Configuration Module *HDWCFG.O* to depend on the Target Parameter File *HDWCFG.UP* and on the AMX header file *CJZZZK.DEF*, a duplicate of file *CJnnnK.DEF*. The make directive to create *HDWCFG.O* consists of two statements, one to make the source file *HDWCFG.S* and one to assemble it.

The Target Configuration Module source file *HDWCFG.S* for use with toolset *XX* can be generated using the AMX Configuration Generator as follows:

```
$(OSPATH)\CFGBLDW\CJnnnCG.EXE $(YOURSRC)\HDWCFG.UP  
$(OSPATH)\TOOLXX\CFG\CJnnnHDW.CT $(YOURSRC)\HDWCFG.S
```

The Configuration Generator *CJnnnCG.EXE* combines the information from your Target Parameter File *HDWCFG.UP* with the AMX Target Configuration Template File *CJnnnHDW.CT* to produce file *HDWCFG.S*.

Your make specification file must also include the directive to assemble source module *HDWCFG.S* as described in the AMX Tool Guide for toolset *XX*.

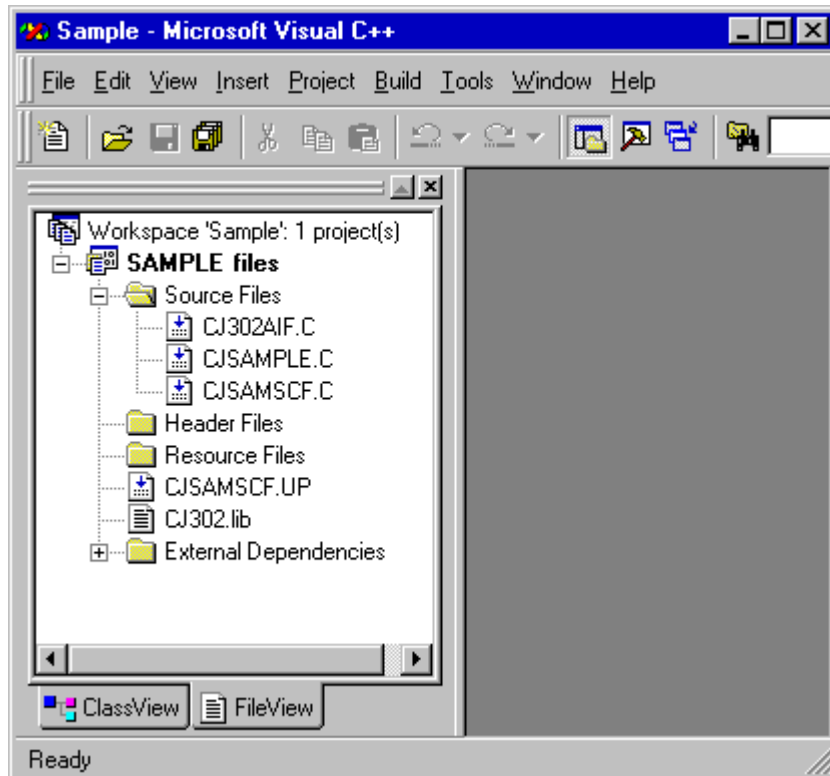
### 3.4 Using the AMX Prototyping System (TAPS)

The AMX Prototyping System (TAPS) can be used to build and test much of your AMX application on a Windows workstation using the Microsoft Visual Studio. Simply follow the step by step instructions presented in the TAPS User's Guide. Within the guide, the Visual Studio project for the AMX Sample Program provided with TAPS is used to illustrate the process.

You can access the manual directly from the Windows Start menu via the link in the TAPS program group. Alternatively, you can view the manual by using your Windows browser to open the installed TAPS file `TAPS302\MANUAL\TAPS302.HTM`.

When you create an AMX application for use under TAPS, you will not require an AMX Target Configuration Module since the target hardware is assumed to be unavailable. You will still need your AMX System Configuration Module.

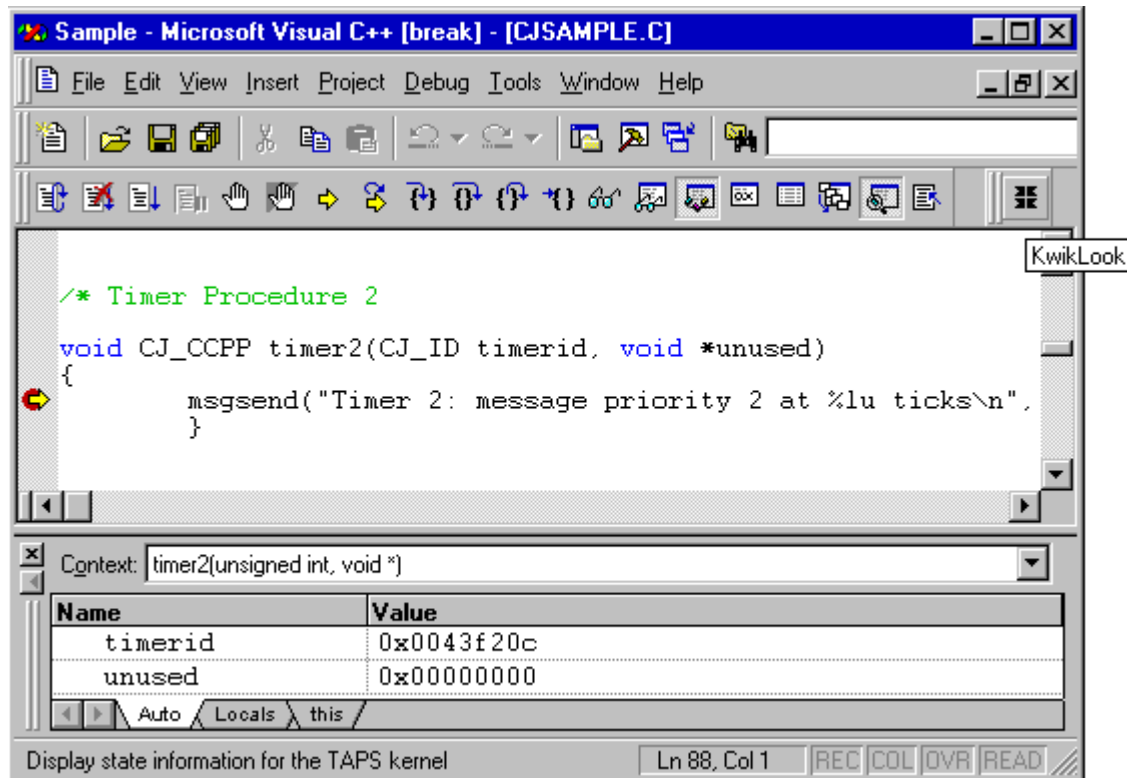
Note that you must use the copy of the AMX Configuration Manager provided with TAPS to edit your AMX User Parameter File. The manager must be run outside the Visual Studio project. The edited parameter file must be added to the project file list, as illustrated by file `SAMPLESCF.UP` in the screen shot below. The project must use the TAPS System Configuration Template File to generate the System Configuration Module `SAMPLESCF.C` which can then be compiled and linked with your application.



## Debugging With TAPS

The Visual Studio Debugger is used to test your AMX application running under TAPS. Start the debugger and run to a breakpoint in the `main()` program. You can then proceed to debug your application using all of the features available to you in the normal Visual Studio debugging environment.

TAPS includes its own variant of the *KwikLook* Fault Finder giving you full task-awareness when testing your AMX application. *KwikLook* is invoked from the custom icon on the Visual Debugger toolbar. You use *KwikLook* as described in Chapter 4.1, just as though you were using it with your target hardware debugger.



This page left blank intentionally.

## 4. Debugging an AMX Application

### 4.1 Using the *KwikLook* Fault Finder

The *KwikLook* Fault Finder is a Windows utility for testing real-time embedded systems developed using KADAK's AMX multitasking kernel. *KwikLook* gives you quick fingertip access to everything controlled by AMX and its managers:

Tasks	Semaphores
Timers (ticks and ms)	Event groups
Mailboxes and message exchanges	Buffer pools
Message contents	Memory pools

All messages queued in mailboxes or message exchanges are visible along with the sender's identification. If no messages are present, *KwikLook* shows the tasks, if any, which are waiting for a message to arrive.

*KwikLook* shows which tasks own resources and which ones are waiting for them. The state of all events in each event group can be viewed complete with a list of the tasks waiting for specific event combinations.

The memory usage display gives a snapshot of current allocation by the Memory Manager. Unexpected fragmentation is readily observable. Similarly, free and used buffers are shown for each buffer pool.

Using *KwikLook*, there are no surprises, no guessing. Disappearing resources can be uncovered. Unexpected task activity is exposed. Even stack overflow and underflow can be observed.

#### Method of Use

The *KwikLook* Fault Finder adds true AMX task-awareness to a variety of popular Windows based debuggers. Your target AMX system must be connected to a conventional host PC running Windows. *KwikLook* access to your AMX application is then provided by your task-aware remote debugger, a Windows program running on the host PC.

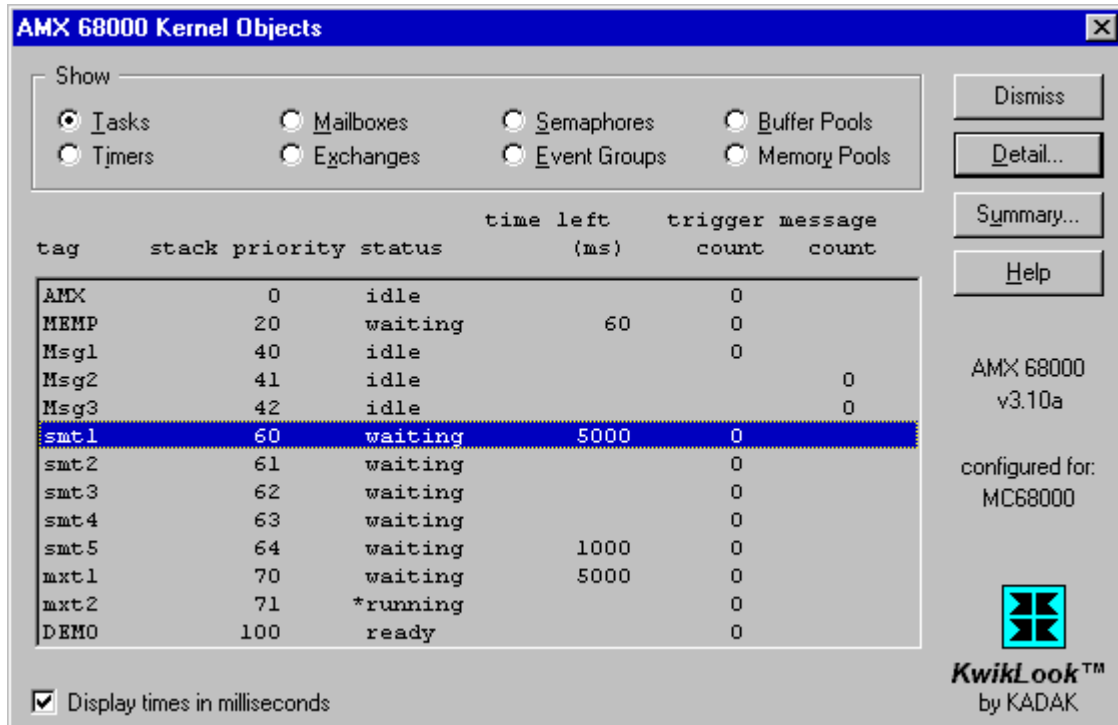
The *KwikLook* Fault Finder is implemented as a Windows DLL which is attached to your debugger when the debugger is started. *KwikLook* is activated by you using a toolbar icon or pull-down menu item provided by the debugger.

## KwikLook Displays

The *KwikLook* Fault Finder is run from the toolbar icon or pull-down menu item provided by your task-aware debugger. *KwikLook* can only be activated once AMX has been started and your application has been stopped at a debugger breakpoint. Once running, *KwikLook* fetches and displays the state of your AMX application in the display window as illustrated below.

At any time, a context sensitive explanation of the display content can be accessed via the Help button. When finished reviewing the state of your AMX application, press the Dismiss button or close the *KwikLook* window to return to the task-aware debugger.

Specific AMX object information is accessed via the radio button selections. The Detail... button can then be used to fetch even more information about the particular AMX object selected from the list.



The screenshot shows the 'AMX 68000 Kernel Objects' window. It features a 'Show' section with radio buttons for Tasks, Mailboxes, Semaphores, Buffer Pools, Timers, Exchanges, Event Groups, and Memory Pools. A table lists kernel objects with columns for tag, stack, priority, status, time left (ms), trigger count, and message count. The 'smt1' object is highlighted. On the right, there are buttons for Dismiss, Detail..., Summary..., and Help. Below these buttons, it says 'AMX 68000 v3.10a' and 'configured for: MC68000'. At the bottom right is the KwikLook logo and 'KwikLook™ by KADAK'. A checkbox at the bottom left is checked and labeled 'Display times in milliseconds'.

tag	stack	priority	status	time left (ms)	trigger count	message count
AMX		0	idle		0	
MEMP		20	waiting	60	0	
Msg1		40	idle		0	
Msg2		41	idle			0
Msg3		42	idle			0
smt1		60	waiting	5000	0	
smt2		61	waiting		0	
smt3		62	waiting		0	
smt4		63	waiting		0	
smt5		64	waiting	1000	0	
mxt1		70	waiting	5000	0	
mxt2		71	*running		0	
DEMO		100	ready		0	

The *KwikLook* Summary... button provides access to the System Summary shown below.

The System Summary dialog box is titled "System Summary" and contains the following information:

- Kernel State:**
  - Run Level: task
  - Task Switch: enabled
  - Time Slicing: disabled
  - Running Task: mxt2
  - Priority: 71
- System Clock:**
  - Current Tick: 0
  - Next Significant Tick: 1
  - Date: 01:00:00 1995/12/01 Fri
- Interrupt Stack:**
  - Top: 0042B198
  - Bottom: 0042B094
  - Size: 256
- Kernel Objects:**

	Used	Max		Used	Max
Tasks	11	of 24	Semaphores	3	of 9
Timers	2	of 10	Event Groups	5	of 10
Mailboxes	2	of 8	Buffer Pools	2	of 6
Message Exchanges	6	of 13	Memory Pools	2	of 5
Message Envelopes	4	of 128	124 free	0 missing	

Buttons: Dismiss, Help

## 4.2 Breakpoints and Tracing

When using a debugger on your AMX system, it is important to be aware of subtle effects which may occur.

When you hit a debug breakpoint, AMX is completely unaware that your debugger is actually executing in the context of the task in which the breakpoint occurred. Fortunately, with most debuggers there is no problem. The debugger usually inhibits external interrupts and switches to a private debugger stack. When you trace or proceed from the breakpoint, the debugger restores the task's stack and resumes execution with interrupts restored.

Debuggers which operate this way can easily be used to debug an AMX application. Just remember that real time stops when a breakpoint is encountered. It only resumes when you allow your system to free-run again. When you proceed from a breakpoint, all device interrupts which have gone pending while at the breakpoint will suddenly generate a flurry of ISP activity with possible task switching side effects.

If you trace single instructions (not whole C statements), the debugger may never actually give up control of the processor. Therefore, while you are single stepping past a breakpoint, your AMX system may remain temporarily shut off because interrupts are disabled.

If you allow your system to run with another breakpoint set several instructions or statements beyond the first breakpoint, it is possible that you may never hit your second breakpoint in the task you are debugging.

For example, suppose that you trace over a call to send a message to a mailbox. If there is a higher priority task waiting at that mailbox for a message, AMX will immediately suspend the task you are debugging and allow the other higher priority task to resume. If the higher priority task reaches a body of untested code and crashes, you may never hit your second breakpoint in the task which you are testing. As long as you are aware of this property, your debugging should proceed smoothly.

Tracing through reentrant code shared by several tasks can be very difficult. When you set a breakpoint in such a procedure, the breakpoint will be hit by the first task to call the procedure. That task may not be the task of interest. Furthermore, when you try to proceed to another breakpoint in the same procedure, you may find that when you hit the breakpoint, you are running in the context of a higher priority task that preempted the first task and called the shared procedure.

### 4.3 Debugging the Launch

Many first time AMX users are frustrated by the inability to locate bugs in their startup code, Restart Procedures or AMX Configuration Module which preclude a successful AMX launch.

Startup is no-man's land. It is a gray area in which your startup code (or your loader) has started AMX but a solid AMX operating environment has not yet been established.

When AMX is executing Restart Procedures, it is in an intermediate state with no user task yet running. Therefore, operations which tasks can perform are not yet acceptable.

You can usually use your debugger to step through your own Restart Procedures although there is no such guarantee. Do not try to step through the AMX procedures which your Restart Procedures call. Once your last Restart Procedure has been called, you must let AMX free run. Any attempt to breakpoint your way through the remaining AMX startup code will almost certainly fail.

If you get through your Restart Procedures and they appear to have worked (i.e. AMX calls did not return error indications and your code only touched devices and data for which it is responsible), then your AMX launch should work. If it does not, the most probable fault is one of the following:

- AMX took a fatal exit and unconditionally halted (see the next topic regarding Fatal Exit Procedures).
- Your AMX Configuration Module contains invalid or unresolved information which leads to improper AMX operation. (This will be unlikely if you used the Configuration Manager and Generator to create your module.)
- You failed to include an AMX option in your configuration which is vital to AMX success. For example, you expect to use AMX timing features but you have not included a clock ISP of any kind.
- Your Restart Procedures caused the AMX Kernel Stack to overflow. You must not sprinkle *printf* statements in Restart Procedures for testing purposes.
- You started AMX with your processor interrupt system disabled but you also had an outstanding, unserviced interrupt pending at that time. Since AMX enables interrupts before calling your Restart Procedures, the device interrupt will be acknowledged by the processor before you even have a chance to install its AMX interrupt handler. Your system will probably fail without the handler needed to service the device.
- You started a device which produces an interrupt but a tested device ISP has not yet been provided to service the device.
- You started an interval timer which expired and caused AMX to execute an untested Timer Procedure.
- You created an interval timer but never started it and therefore your Timer Procedure is never executed.
- You created a task but never triggered it or sent it a message and it therefore never executes.

Look to your Restart Procedures and your AMX Configuration Module for the source of your startup problems. No startup errors have yet been traced to AMX. (It doesn't rule out AMX; it just makes it unlikely.) Many startup problems have eventually been traced to modifications made to pieces of AMX test program code "borrowed" and adapted for a new application.

### **Using a Fatal Exit Procedure**

Do not ignore the use of the AMX Fatal Exit Procedure as a very powerful debugging tool. If your System Configuration Module contains anomalies which preclude proper AMX operation, AMX may abort a launch and take a fatal exit.

If you have not modified AMX Fatal Exit Procedure *cjksfatal* in module *CJnnnUF.C* to suit your needs, AMX will halt with interrupts disabled forcing you to initiate a power reset to recover. However, you can intercept this fatal shutdown by revising the AMX Fatal Exit Procedure which, although very restricted in what it can do, can at the very least give you an indication that the fault has occurred. Read Chapter 14.1 of the AMX User's Guide for the rules.

## 4.4 Debugging Caveats

If the debugger does not switch to a private stack, it may use more stack than has been provided for the task in which the breakpoint occurred. The debugger will therefore probably crash in the AMX task or at least force a crash to occur when you proceed from the breakpoint.

If the debugger executes with interrupts enabled, strange effects may be noticed. If the debugger's stack is too small to meet AMX task stack specifications, the debugger will probably crash with the first interrupt that occurs after the breakpoint.

Even if the debugger stack is adequate, strange effects may occur. Since interrupts are enabled, all interrupt driven activity continues to occur while the debugger is stopped at the breakpoint awaiting your instructions. If, as a consequence of interrupt activity, a task of higher priority than the breakpointed task becomes ready to run, AMX will perform a task switch. The higher priority task will run and your debugger will temporarily disappear until the higher priority task completes or becomes blocked again.

This disruption of the debugger's operation may be enough to cause some remote debuggers to lose communication with their host computer and appear to crash.

Never use your debugger's *QUIT* command to leave your AMX system. Your AMX system must invoke *cjksleave* to force an orderly AMX shutdown. When AMX attempts to return to your *main()* program, the debugger will indicate that your program under test has terminated. Only then can you use your debugger's *QUIT* command to terminate the debug session.

Never use a debugger's command (such as *ctrl-c*) to try to stop a "run-away" AMX system. This mechanism is not compatible with your multitasking environment and often leads to catastrophic failure. The debugger must only gain control via breakpoints, watchpoints or traces.

You may find source file path information embedded in object and library modules to be very aggravating when you move an application to a different machine for testing. You may find that your debugger cannot locate the source code for the module which you are testing because the path used to compile the module does not exist on the test machine.

### Undefined AMX Procedures

When debugging an AMX application, you may observe that some AMX procedure names appear to be missing from your symbol table. These AMX procedures are mapped directly to other equivalent procedures by macro definitions in the AMX header files.

For example, the AMX Resource Manager procedures map directly to AMX Semaphore Manager procedures. In this particular case, you can override the mapping and force an actual set of Resource Manager procedures to be loaded by editing file *CJnnnAPP.H* to include the following statement (see Appendix D.2 of the AMX User's Guide). Recompile all application modules which reference Resource Manager procedures.

```
#define CJ_OPTRM
```

## Extraneous Interrupts at Startup

As indicated in Chapter 4.3, strange things can happen if you inhibit device interrupts and then launch AMX with an unacknowledged device interrupt pending. The interrupt will be acknowledged by the processor as soon as AMX enables interrupts, just before it calls your first Restart Procedure. Consequently, the interrupt will occur with unpredictable and often catastrophic results before you even have a chance to install a suitable AMX device interrupt handler.

The unexpected interrupt is frequently a clock interrupt left pending from a previous debugging session. For example, suppose that you successfully load and start debugging your AMX application. You reach a breakpoint but realize that you have gone too far. You use your debugger to reset your system, possibly even reloading your program and then start a new debug session. Unfortunately, your debugger may not totally mimic the effects of a hardware reset. Hence, a clock interrupt may still be pending from your prior debug session, even though it is masked off by the debugger. When the debugger starts your program for the second time, you may get a clock interrupt the instant the debugger enables interrupts. The AMX clock interrupt handler which was installed during your first debug session will respond to the interrupt and try to service the clock, completely unaware that AMX is no longer even in the picture. As expected, the result is chaos.

Take another example. Suppose that you successfully load and start debugging your AMX application, only to detect a problem in the coding of one of your tasks. You exit from your debugger, repair the fault, rebuild your system and start a new debug session. Unfortunately, you may have had a device interrupt pending at the time that you stopped your first debug session. The debugger may have masked off your device interrupt, leaving you blissfully unaware of the pending doom. When your application starts the second time, bang! You get an interrupt from a device that you thought was reset.

Often your debugger is the source of the unexpected interrupt. Consider a debugger that uses a serial port or Ethernet connection to your target hardware to load and execute your AMX application. Assume that a debug monitor in the target uses an interrupt driven device driver to communicate with the debugger. The debugger finishes downloading your application and instructs the monitor to start your program. If the monitor simply inhibits the device's interrupt request at the processor's interrupt controller without first resetting or disabling the device, you may encounter an interrupt from the device when you least expect it. For example, such an interrupt could occur if your application updated the interrupt controller's interrupt mask and inadvertently enabled the specific device interrupt.

## 5. AMX Programming Hints

### 5.1 Application Portability

If you are coding in C and expect to port your AMX application to a different processor, observe the following portability rules.

Include the generic AMX header file *CJZZZ.H* in your application C source modules. Then, to recompile a source module for a different target processor, you simply use the generic AMX header file *CJZZZ.H* provided with AMX for that target. Editing of every source module will not be required.

Make all AMX task and object identifiers be of type *CJ\_ID*.

Make all AMX timer values be of type *CJ\_TIME*.

Use only the least significant 16 event flags of each event group. This will permit designs for 32-bit processors to be readily ported to 16-bit processors.

Do not use unions to extract *char* or *short int* values from *long* or pointer variables. The byte reversal of little endian versus big endian products will kill you.

If you use the keyword *CJ\_CCPP* to define the parameter passing rules when declaring public procedures, it will ease porting C code from one C compiler to another.

Use the AMX *cjcfvolXXXX* procedures to fetch volatile variables so that porting code to C compilers that do not support the keyword *volatile* will be possible.

Use the identifier *CJ\_NULL* for a *NULL* data pointer and *CJ\_NULLFN* for a *NULL* function pointer so that your code will easily port to processors on which data and code must be treated differently.

Align AMX messages on 32-bit boundaries to improve execution speed and to ease porting your application to 32-bit processors.

## 5.2 AMX Stack Allocation

Each AMX task requires a separate storage region for use as a task stack. AMX also allocates a Task Control Block for the task from the storage provided. These stacks must be large enough to accommodate the deepest possible level of procedure nesting. Use the following rules to calculate stack sizes:

1. Start with the minimum task stack size required by AMX. This stack is used to store the task state when it is interrupted. The minimum stack allows a task procedure with no local variables to call any AMX procedure and return to AMX.
2. Determine the stack size for each procedure called by the task. Most C procedures will require a minimum of eight bytes for storage of return address and saved registers plus storage for all calling parameters and automatic variables. On RISC processors, the stack frame requirements can be quite large.
3. Find the procedure nesting path that requires the most stack space and add the stack sizes of each procedure to the minimum task stack. Recursive procedures must have a stack size large enough to allow for the maximum recursion depth.

The stack provided by your C's startup module will only be used by your *main()* program as it starts your AMX system. This stack is only used by AMX during the launch and after a shutdown.

### Stack Checking

Some C compilers generate a runtime stack check at the entry point to every C procedure. The check verifies that, after local variable storage has been allocated on the stack, stack still remains available for use.

This stack checking is usually a compilation option which can be inhibited with a switch during compilation. Unfortunately, you may find that the C Runtime Library is delivered with its modules compiled with stack checking enabled. Hence, if your program requires these runtime library procedures, you will have stack checking in effect.

The AMX Tool Guide for the toolset which you are using will provide instructions, if necessary, for defeating the compiler's stack checking.

## Stack and Data Alignment

By design, all AMX data structures are 32-bit aligned to ensure optimal performance on processors which support 32-bit access to memory.

AMX also ensures that stack alignment is maintained throughout all AMX code paths. However, the AMX design intent can only be achieved if the private AMX data region and all AMX and application stacks are properly aligned to begin with. Most 32-bit processors require stacks to be 32-bit aligned. However, there are processors which require 64-bit and even 128-bit alignment. When coding functions in assembly language, be sure to adhere to your processor's stack and data alignment specifications.

Typically it is your program linker which ensures proper alignment. In some cases, you may have to use a linker directive to specify the alignment to be used for specific sections (code, data, etc.) from each module in your link. All data and stacks in the AMX System Configuration Module will then be properly aligned.

Failure to meet the alignment requirements of your hardware can lead to strange timing effects or even run-time faults. For example, while testing AMX on one particular processor, the AMX data region and stacks were inadvertently not 32-bit aligned. Consequently, every 32-bit data and stack access required two memory accesses. Timing measurements indicated that many AMX procedures took approximately 30% longer to execute.

## 5.3 Choosing a Synchronization Method

### Task Wait/Wake

Use the AMX task wait/wake mechanism to synchronize tasks to simple, one of a kind, slowly occurring events in ISPs or other tasks. It is most suitable for events occurring at 100 Hz or slower rates on 10 MHz processors. If timeout is required, use procedure *cjtkwaitm*.

A task can use the pending wake feature as follows to guard against losing events. The task calls *cjtkwaitclr* to clear any pending wake condition. Then the task initiates the action that will produce the event of interest. The task then calls *cjtkwait* to wait for the event. If the event occurs before the task can enter the wait state, the task will continue to run without waiting because of the pending wake posted by the event's *cjtkwake* call.

The task id can be used as a convenient boolean indicator. Create an event variable of type *CJ\_ID* and set it to *CJ\_IDNULL* to indicate that no task is waiting for the particular event. When a task is about to wait for the event, it can set the event variable to the task's id (using *cjtkid*) thereby informing the event handler that a task is waiting and identifying the task at the same time.

### Task Trigger

Use task triggers for rapidly occurring events in which event counts are significant and little, if any, information is required by the task to service the event. A Restart Procedure initiates the action that will produce the events of interest. The event handler calls *cjtktrigger* to trigger the task which will service the event. The task executes once to completion for each trigger.

Use a circular list to pass 8, 16 or 32 bits of information to the task. The event handler adds the information to the bottom of the list and the task retrieves the parameters from the top of the list. Larger amounts of data can be handled by using buffers from a buffer pool. The event handler can pass the buffer pointer on the list and the task can release the buffer when it completes processing the information in it.

### Counting Semaphore

A counting semaphore can be used exactly like the task trigger mechanism just described. However, there is more task switching overhead required since the task state must be saved and restored when a task waits for a semaphore.

Unlike a task trigger, the counting semaphore gives the task greater flexibility in determining when and where within its code sequence the event wait should take place.

A task creates a counting semaphore with an initial count of 0. Then the task initiates the action that will produce the events of interest. The task then calls *cjasmwait* repetitively to wait for the events.

A counting semaphore can also be used when any one of several tasks must be synchronized to an event. For example, assume that two server tasks are available to handle events and it does not matter which of the servers handles a particular event. Each of the server tasks can wait on a single counting semaphore which is signaled by the event handler. The server tasks respond to events in FIFO fashion.

### **Resource Semaphore**

Always use a resource semaphore to control access to anything like numeric coprocessors, non-reentrant libraries, data base records or disk files. The resource semaphore provides the necessary characteristic of ownership.

The resource semaphore provides the additional benefit of allowing nested ownership. Consequently, a task which owns a resource can successfully call application procedures which unknowingly try to reserve the same resource. Since the task calling the procedure already owns the resource, the procedure is allowed to execute without being blocked as would otherwise occur if a simple binary counting semaphore had been used to control access to the resource.

### **Mailbox or Message Exchange**

The AMX mailbox and message exchange offer information passing synchronization methods. The event handler creates a message which it sends to a mailbox or message exchange. Any task can ask for or wait for a message from the mailbox or message exchange, thereby synchronizing with the event handler which generated the message.

A task, having received a message, knows that a particular event has occurred and has a complete description of the event in the task's copy of the message generated by the event handler.

A great deal of flexibility is provided by this method of synchronization. As with most AMX synchronization methods, you can control the order in which tasks queue waiting for messages. In the case of a message exchange, you can also control the order of priority in which messages are sorted at the exchange, thereby reordering the sequence in which the events are actually serviced.

### **Message Exchange Task**

You can use a message exchange task for synchronization in much the same fashion as a trigger task. The event handler creates a message which it sends to the message exchange task's private message exchange. There is no need to trigger such a task. The task automatically receives the message, processes it and ends, ready to receive the next message when it becomes available.

## Ack-Back Messages

Use the AMX ack-back facility to avoid the need for extra task to task synchronization semaphores. A task can send a message to a mailbox or message exchange at which another task has agreed to rendezvous. The sending task waits for its message to be delivered and acted upon. The receiving task processes the message and acknowledges its receipt, allowing the sending task to resume execution.

A message exchange task's private message exchange can also be used for this type of synchronization.

## Event Group Flags

Use event flags strictly for handling asynchronous, combinatorial event logic. Also use event flags when multiple tasks must be concurrently synchronized to exactly the same event condition(s).

Each event flag in an event group should be altered by one, and only one, event handler. Abiding by this restriction ensures that state driven flags always match the actual event condition and that pulsed event flags are only pulsed when the event occurs.

Event processing by the Event Manager is inherently slower than other synchronizing methods because races among sequentially occurring events must be resolved sequentially to ensure that specific event combinations are always detected in the order in which they occur. The Event Manager uses the AMX Kernel Task to resolve such races.

Event flags can still be attached to high-speed interrupt driven events without compromising interrupt response. Race resolution is deferred by the AMX Interrupt Supervisor to the Kernel Task.

This extra switch to the Kernel Task, although essential for event race resolution, can and should be avoided by using any of the previously described methods for simple ISP/task synchronization.

### Warning

Do not use event flags unless a task must synchronize to multiple, asynchronous events or multiple tasks must synchronize to the same event.

## 5.4 AMX Caveats

### Task Trigger vs Message Queuing

An AMX message exchange task is a special kind of task with a private message exchange bound to it by AMX as described in Chapter 14.6 of the AMX User's Guide.

You must not call procedure *cjtktrigger* to trigger an AMX message exchange task. A task of this type is automatically triggered by AMX when it is bound to its private message exchange by your call to procedure *cjtkmxinit*.

A basic AMX task executes once each time it is triggered. The task receives no parameters when it runs.

By contrast, the AMX message exchange task is triggered once, and only once, by AMX. Thereafter, the task procedure is called once each time a message arrives in the task's message exchange. The message is passed to the task by value or by reference as a parameter to the task procedure.

### Message Envelopes

AMX uses message envelopes for passing messages to mailboxes and to message exchanges. However, AMX also uses message envelopes for passing messages to the AMX Kernel Task. Kernel messages are generated if a task or ISP must defer an operation to the Kernel Task in order to resolve an otherwise disastrous race condition. You must always provide some message envelopes for use by AMX. A minimum of ten (10) envelopes is recommended.

### AMX Message Length

AMX messages originate as user defined blocks of 12 or more sequential bytes of memory. The maximum length ( $n \geq 12$ ) 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.

Whenever you send a message to a mailbox or to a message exchange, you point to memory containing the message. AMX copies all  $n$  bytes into an AMX message envelope and attaches the envelope to the appropriate message queue. The sender may be a task, ISP, Timer Procedure, Restart Procedure or Exit Procedure.

When a task gets a message from a mailbox or message exchange, AMX removes the envelope from the message queue and copies all  $n$  bytes from the envelope to the storage area provided by the task. Failure to provide at least  $n$  bytes of alterable storage for the message is a common fault. Remember that AMX will copy  $n$  bytes.

## Common Message Problems

On some 32-bit processors, the AMX message copy may be slow if the message source (or destination) is not 32-bit aligned. On others, a data alignment exception may be generated. The following examples illustrate BAD coding techniques which may lead to slow execution or faults at runtime.

```
extern CJ_ID mailboxid;

const char messageA[] = "Fixed long msg!";
const struct {
    char    xopcode;
    char    xparameter;
} messageB = {5, 'P'};

void badcode(void)
{
    cjmbSEND(mailboxid, messageA, CJ_NO);
    cjmbSEND(mailboxid, &messageB, CJ_NO);
}
```

The first message, *messageA*, is a constant character array which most C compilers will place in memory at a long aligned address. However, since the array is a character array, the compiler is free to align the array at any byte address if it so desires. If the array *messageA* is not long aligned, the AMX procedure *cjmbSEND* may execute slowly because of the 32-bit access at the improperly aligned address.

A similar problem may exist with *messageB*. Again, most C compilers will place a structure in memory at a long aligned address. However, some compilers will relax the structure alignment to just meet the minimal alignment needs of the structure members. In this example, since all members of structure *messageB* are characters, the compiler is free to locate the structure on any character boundary.

Even if the compiler does long align *messageA*, a problem remains. The whole message string, "Fixed long msg!", will not be sent to the mailbox unless you have increased the AMX message size beyond its default minimum length of 12 bytes. Only the first 12 characters, i.e. "Fixed long m", will be sent in the AMX message. Also note that a trailing null character '\0' will not be present in the AMX message.

## AMX Shutdown

If you use the AMX procedure *cjksleave* to stop your AMX system and return to the point of launch, you must first ensure that all device operations and AMX task activity have come to an orderly halt. The responsibility is yours; AMX does not know anything about your application and how it works.

During the exit process, the AMX task scheduler continues to operate. All AMX managers remain functional.

Since Exit Procedures run in the context of the task which initiated the shutdown by calling *cjksleave*, they are free to use any of the AMX task synchronization methods to wait for other tasks to do their windup processing. Of particular use is the message acknowledgment facility. An Exit Procedure associated with a task having special shutdown responsibilities can send a shutdown message to the task and wait for an ack-back from the task.

Once all of your Exit Procedures have been executed, AMX shuts down the AMX kernel and all of the AMX managers. At that point, if you still have any interrupt activity pending which requires AMX for service, your system will most probably crash.

## Code Size and Speed

Predefine tasks and other AMX objects in your System Configuration Module. It is easier and error free and you will eliminate the extra application code needed to dynamically create these AMX resources.

If you do not need message ordering by priority, use mailboxes only. Message exchanges require more memory and are slightly slower to use. Do not create message exchange tasks.

If you increase the AMX message envelope size, the AMX data area will grow accordingly and AMX message passing will be marginally slower. Do not forget that AMX occasionally uses these envelopes for its own private purposes.

Restrict the number of tasks and other AMX objects to reasonable limits for your application. If your number of tasks exceeds 30, call KADAK for technical support. A well designed application should rarely exceed 15 tasks.

Do not use the C *interrupt* keyword or pragma on any procedure unless you are purposely coding a nonconforming ISP. The AMX ISP root eliminates the need for this non-portable C feature. AMX Interrupt Handlers can be coded as standard C procedures.

Do not use *cjksfind* or *cjksgbfind* in ISPs or Timer Procedures to find ids of AMX resources. These tag lookup procedures are relatively slow. If an ISP or Timer Procedure needs some AMX id, find the id at launch time or in some task and make the id permanently available in a private id variable. Note that the AMX ids of all predefined objects are always available in the id variable provided with the object's definition.

Do not use *cjtmconvert* in ISPs or Timer Procedures to convert milliseconds to AMX ticks. This procedure is relatively slow. Compute the value at launch time or in some task and make the value permanently available in a private variable of type *CJ\_TIME*.

## 5.5 Interrupt Latency

The term interrupt latency is defined in Chapter 4.1 of the AMX Timing Guide. The measured AMX interrupt latency is the longest interval during which AMX inhibits all external interrupts. Specific latency figures are published in the AMX Timing Data sheets for different processors and toolsets.

If interrupt latency is of particular importance in your application, there is one guideline which, if followed, will lead to improved performance. Avoid the use of AMX Scheduler Hooks. These hooks are described in Chapter 14.3 of the AMX User's Guide. The worst case AMX interrupt latency occurs in the path through your hooks into the AMX scheduler.

## 6. C Programming Primer

### 6.1 C Programming Practices

With the exception of your AMX Target Configuration Module, all of your AMX application modules can be coded in C. Tasks, Timer Procedures, Restart Procedures and Exit Procedures are readily coded in C. Interrupt Service Procedures can also be coded in C although assembler is recommended. Some procedures such as AMX Task Scheduler hooks must be coded in assembler.

It is recommended that you thoroughly familiarize yourself with the User's Guide and Library Reference Manual provided with the particular C compiler that you are using.

It is assumed that you are thoroughly familiar with the AMX User's Guide, the AMX Target Guide for the processor of interest and the relevant chapter of the AMX Tool Guide for the particular C compiler that you are using.

The use of C in a multitasking environment poses special difficulties. Some of the more frequently encountered problems are described in this chapter.

#### Procedure Prototyping

The AMX header files include prototypes for all AMX procedures.

If your C compiler does not support prototyping, you will have to edit the AMX header files to remove all formal parameter specifications from the procedure definitions.

#### AMX Typedefs

AMX uses a private handle to identify system objects such as tasks and timers over which it has control. A handle is an unsigned integer identifying a particular object.

The AMX header file *CJnnnCC.H* includes the following C *typedef* of symbol *CJ\_ID*:

```
typedef unsigned int CJ_ID;
```

All AMX identifiers provided to or received from AMX procedures are then declared to be of type *CJ\_ID*. If you code to this convention, your AMX application modules will be more readily portable to versions of AMX for other processors.

The symbol *CJ\_IDNULL* is also defined giving a valid and portable definition of a non-existent id.

## Global and Static Variables

Variables that are defined outside of any C procedure are known as global variables. They may be accessed by any procedure that declares them to be external (*extern*). A global variable resides in a single module (file). When a global variable is initialized to a value, the initialization takes place in the module in which the variable resides.

A global variable can be made private to a module by declaring it to be *static*. A static global variable can be accessed by all of the procedures in the same module but cannot be accessed by any procedure in another module.

C allows global variables to be initialized. For example:

```
int variab = 0x1234;
```

This declaration (outside of any procedure) declares an integer with an initial value of `0x1234`.

Initialized global variables are set to their initial value prior to starting your `main()` program. Uninitialized global variables are set to zero. These initializations may not take place in a ROM system. Refer to Chapter 6.4 for a more detailed discussion of C startup requirements in ROMed systems.

## Atomic Variable References

Global public variables present a particular hazard on processors whose architecture precludes the atomic (indivisible) modification of memory. If two concurrently executing tasks share a common public variable, then modifications of the variable must be atomic. Each task wishing to modify the variable must read, modify and write the variable in one indivisible sequence.

### Suggestion

Eliminate global public variables and watch your system error rate go down.

## 6.2 Structure Packing

Some compilers optimize storage alignment for speed, not for size. Furthermore, 16-bit and 32-bit variables may have to be even aligned to avoid memory access exception traps. Hence, many "gaps" in structures may exist as the compiler forces alignment of fields for proper access on the processor.

You should coerce your C compiler to pack fields within structures which are used as AMX intertask messages. An AMX message may be configured to be an arbitrary block of 12 bytes (minimum) which is passed by value.

For example, on 32-bit processors the following AMX message includes 10 bytes of information.

```
struct {
    char    c1;
    int     i1;
    char    c2;
    long    lv;
} msg;
```

However, if the C compiler forces 32-bit alignment of the integer and long variables in this structure, the structure size will be 16 bytes. The last four bytes of this message will not be transmitted by AMX since the message length exceeds the 12 byte AMX message size assumed in this example.

Most C compilers provide an option to force byte alignment of variables at the expense of marginally slower execution speed. Judicious choice of message structures can also be used to eliminate the problem. For instance, if characters *c1* and *c2* are moved to follow long *lv* in the example, the message length reduces to 10 bytes since gaps are avoided.

If your C compiler will not pack structures, be sure to set your definition of the AMX message size to match your largest AMX message.

## 6.3 Reentrancy and Concurrent Execution

A procedure is reentrant if it executes properly even when it is interrupted and called from the interrupting program. A reentrant procedure may not modify any variables at fixed locations in memory (static or global variables). All variables must be local (automatic variables).

Here is an example of a non-reentrant procedure:

```
/* Return x-squared + x (non-reentrant) */
int poly(int x)
{
    static int tmp;

    tmp = x + 1;
    return(tmp * x);
}
```

This procedure is non-reentrant because *tmp* is declared static. Suppose that this program is calculating the polynomial for  $x = 2$ . The program is interrupted just after the assignment to *tmp* occurs with a value of 3. The interrupting program requires the polynomial to be calculated for  $x = 3$ . It calls the procedure and the value 12 is returned. However, *tmp* is left with the value 4; that is,  $3 + 1$ . When the original program resumes, it will return the value 8 instead of the correct value, 6. This problem may be corrected by declaring *tmp* to be an automatic variable or a register variable.

A reentrant procedure may only call reentrant procedures. Calling non-reentrant procedures will make it non-reentrant.

A routine need only be reentrant if it is called by more than one concurrently executing procedure. For instance, tasks need not be reentrant but a routine called by more than one task must be reentrant because tasks can execute concurrently.

### Warning

Some procedures in your C Runtime Library may not be reentrant. This is especially true of many transcendental math procedures and some floating point procedures, especially those which use a numeric coprocessor.

Concurrent execution within an AMX system is defined by the following rules:

1. Tasks execute concurrently with each other, with Interrupt Service Procedures (ISPs), with Timer Procedures and with Exit Procedures. Tasks do not execute concurrently with Restart Procedures.
2. Interrupt Service Procedures execute concurrently with tasks, with Timer Procedures, with Exit Procedures and, in the case of nested interrupts, with other ISPs. They may execute concurrently with Restart Procedures if you start your devices in a Restart Procedure.
3. Timer Procedures execute concurrently with tasks, with ISPs and with Exit Procedures. They do not execute concurrently with each other or with Restart Procedures.
4. Restart Procedures may execute concurrently with ISPs if you start your devices in a Restart Procedure. Restart Procedures do not execute concurrently with each other or with any other procedure.
5. Exit Procedures execute concurrently with tasks, with Timer Procedures and with ISPs. They do not execute concurrently with each other or with Restart Procedures.

## 6.4 Using the C Runtime Library

Not all procedures in your C Runtime Library are reentrant. Floating point and math routines are often not reentrant. File handling, I/O and memory allocation procedures (*fopen*, *fgets*, *printf*, *malloc*, *calloc*, *free*, *sbrk*, *scanf*, etc.) are, in general, not reentrant.

You must treat the error variable *errno* (and all variables like it) as meaningless. If a task is preempted before it can read *errno*, the value of *errno* may be altered by C library procedures called by higher priority tasks.

String conversion procedures such as *strtok* are not reentrant because they maintain private pointers for use on subsequent calls. Most simple string manipulation and data conversion procedures (*strcpy*, *strcmp*, *atoi*, *isalpha*, etc.) are reentrant and may be used safely anywhere. If in doubt, assume the worst.

When calling non-reentrant procedures in a multitasking environment, you must protect the code to avoid problems. One approach is to ensure, by design, that only one task at a time ever calls the procedure.

If this approach is too restrictive, the AMX Semaphore Manager can be used to lock the C Runtime Library whenever non-reentrant library procedures must be invoked. Treat the library as a resource using a resource semaphore and then reserve the library prior to any call to a non-reentrant library procedure. When you are finished using the library, be sure to release the library for use by other tasks.

### Memory Management and *malloc*

For most processors with a flat memory model, the C memory allocation procedures will work properly. However, tasks cannot concurrently share *malloc*, *free*, etc. (or procedures which call them) unless you treat such procedures as resources as described above.

On processors with segmented memory architectures, the C memory allocation procedures may fail. C libraries which assume that the memory heap is located immediately above the stack segment will fail when using AMX because every task has its own private stack segment.

Use the AMX Memory Manager if you require dynamic memory allocation. Have your *main()* program call *malloc* to allocate the largest available region of memory possible. Save the pointer to this region for use as an AMX memory section. Then, in a Restart Procedure, create a memory pool and assign the memory section to it. Tasks can then dynamically allocate memory from this memory pool.

### Using Third-Party Libraries

Many AMX applications require your use of specialized third-party libraries for database access, screen access, graphics, etc. These library packages are often not reentrant and hence not sharable simultaneously by AMX tasks. They may be used in your AMX system provided some form of protection is provided. Use a resource semaphore to reserve the package while it is in use.

## C Startup Code and ROM

Contrary to what some may say, most C compilers can be used to produce ROMable code. The following guidelines may not be applicable to all AMX users. Much depends upon the constraints of your application.

Initialized and uninitialized static data presents the biggest problem with ROMed code. The C language includes no built-in features to simplify the problem of initializing static variables which must exist in RAM with values that must come from ROM. When static initialized data is defined in a C module it just becomes part of the data section.

Constants may also be placed in a data section and hence may not automatically be part of your ROM image. You should also be aware that some C code generators occasionally optimize code by creating private constants which are placed in the initialized data section. This is especially prevalent on processors such as the Intel 80x86 with a segmented memory architecture.

String constants may also be placed in the initialized data section instead of in the code or constant section. C compilers do this to meet the C language specification which permits such strings to be altered at run time.

In order to embed your application in ROM, you will require a link and locate utility which can place code, constant data, initialized data and uninitialized data into separate regions of memory. A copy of the initialized data region must be physically present somewhere in the ROM image but at an address distinct from its actual runtime location. The C startup code must then copy the initialized data from the ROM to the required runtime RAM location prior to calling your *main()* program.

## 6.5 Bootstrap and C Startup Code

Every target board and toolset has unique power on, processor setup, board initialization and C startup requirements. Before AMX is launched, the unique hardware initialization for the target board and software initialization for the C runtime environment must have been done.

For some toolsets, the C startup code provided with the toolset *xx* compiler is inadequate for use in embedded AMX applications. In such cases, AMX includes a replacement startup module which you can tailor to your needs. To see if such a replacement module is needed, refer to the file list in the AMX Sample Program link/locate file for toolset *xx* in the board specific directory of AMX installation subdirectory *TOOLXX\SAMPLE*.

The C startup code must copy all initialized data from the program's ROM section to the required runtime RAM location prior to calling your *main()* program. It is the C startup code which also sets all uninitialized data to zero prior to calling *main()*.

Most tool vendors include the source for the C startup code so you can change it to meet your needs. If you choose to provide your own startup code, rename your *main()* function and call the revised function from your C startup code. Otherwise, the mere presence of a function with name *main()* may force your linker to load the C library startup code, defeating your attempt to replace it.

Note that if you omit the C startup code, some C library procedures (especially those for device I/O and math operations) may not be usable since you may have eliminated their internal initialization. Many ROM based systems are unaffected by this constraint since they require no general purpose device or floating point support.

If you replace the C startup code with your own implementation, it becomes your responsibility to initialize the data regions of memory as described above.