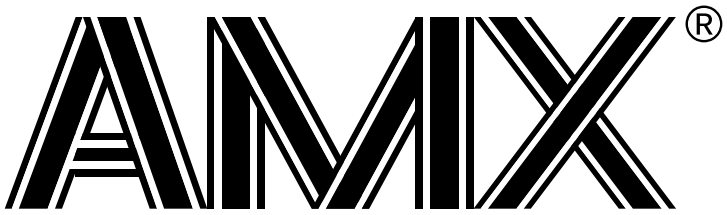

AMX™ 86 Tool Guide

First Printing: **November 1, 1990**
Last Printing: **March 1, 2005**

Copyright © 1990 - 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

Copyright © 1990-2005 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.

AMX 86 TOOL GUIDE

Table of Contents

Page

1. Selecting a Tool Set

Available Toolsets

AMX™ 86, the *KwikLook*™ Fault Finder and the InSight™ Debug Tool have been developed on a PC with Microsoft® Windows® using the software development tools described in this guide.

To simplify the selection process, KADAK has prepared this Tool Guide. This chapter introduces the tools and defines the subsets which KADAK has used with success. Subsequent chapters provide specific guidelines for using each of the supported toolset combinations with AMX 86.

Note that AMX 86 is delivered to you ready to use with each of the supported toolsets. Should you wish to rebuild AMX 86 for any reason, follow the construction guidelines provided in Appendix A.

To construct your embedded application, you will require a C or C++ compiler, an assembler, a librarian (optional), a linker and a remote debugger. The vendors listed below provide these tools. The tool name listed is the vendor's product name or the name of the executable program used to run the tool. The tool name listed will be used throughout this manual to reference the specific tool from a particular vendor.

Vendor	C/C++	Assembler	Librarian	Linker	Locator	Debugger
Paradigm C/C++	*PCC*	*PASM*	*PLIB*	*PLINK*	*LOCATE*	*via IDE*
Microsoft Visual C/C++	*CL*	*MASM*	*LIB*	*LINK*		*CodeView*
Borland C/C++	*BCC*	*TASM*	*TLIB*	*TLINK*		*Turbo Debug*
WATCOM C/C++	*WCC*	*WASM*	*WLIB*	*WLINK*		*WD Debugger*

Supported Toolsets

Unfortunately you cannot arbitrarily use any combination of the listed tools. Of all the tools listed, KADAK has identified several combinations which can be used with AMX 86. The supported toolsets are divided into major classes according to the C/C++ compiler vendor.

Each supported toolset is given a two character mnemonic called a **toolset id** which is used by KADAK to identify the toolset combination. The two characters of the mnemonic identify the compiler vendor. A third character, if needed, identifies the locator and/or debugger used.

	Compiler
PD	Paradigm Systems 16-Bit C/C++ (for 16-bit memory access)
PX	Paradigm Systems 16-Bit C/C++ (for 24-bit memory access)
MC	Microsoft 16-Bit Visual C/C++
TC	Inprise, Inc (formerly Borland) C/C++
WC	Sybase, Inc. (formerly WATCOM) C/C++

The following toolset combinations are supported by KADAK.

Toolset id:	**PD, PX**	**TC**	**MC**	**WC**
Vendor:	Paradigm	Borland	Microsoft	WATCOM
C/C++	*PCC*	*BCC*	*CL*	*WCC*
Assembler	*PASM*	*TASM*	*MASM*	*MASM*
Librarian	*PLIB*	*TLIB*	*LIB*	*WLIB*
Linker/ Locator	*PLINK LOCATE*	*TLINK*	*LINK*	*WLINK*
Debugger	*via IDE*	*TD*	*CV*	*WD*

Note that the Microsoft *MASM* assembler is used for the WATCOM *WC* toolset because of flaws in the WATCOM *WASM* v10.5 assembler.

DOS vs Embedded Systems

The Microsoft, Borland and WATCOM toolsets link your AMX application to produce a DOS executable *EXE* load module. Use one of these toolsets if you intend to create an AMX 86 application which must be started from DOS, even if the application does not make use of DOS or BIOS services.

For a truly embedded system in which your AMX 86 application is either committed to ROM or downloaded for execution by some other means, use the **Paradigm Systems** toolset. This toolset supports the linking and locating of your AMX application to meet your target hardware memory requirements. It gives you full control over the placement of code and data into ROM and RAM. Furthermore, Paradigm provides the bootstrap code and custom C startup code necessary to meet the needs of embedded developers.

The Paradigm tools create a linked *EXE* file which is then converted into a form suitable for ROMing. Paradigm's debugger, adapted from Borland's Turbo Debugger, is a source level debugger with remote debugging services. The Paradigm debugger can be used with KADAK's *KwikLook* Fault Finder for task-aware debugging of your AMX 86 application.

The Paradigm tools can be used to develop applications for conventional real-mode, 80x86 target hardware which employ 16-bit memory addressing. However, the Paradigm tools also support the 24-bit addressing introduced by VAutomation with its Turbo86 line of processors. Use Paradigm toolset PD for 16-bit systems. Use Paradigm toolset PX for 24-bit systems.

Eight Steps to Success

The sheer volume of detail provided with AMX may at first be daunting. However, constructing an AMX application is actually a very simple eight step process made even simpler by the AMX Configuration Manager, a Windows® utility provided with AMX.

1. If you are using the PC Supervisor, go to step 2. Otherwise, select the AMX Clock Driver which most closely matches your target hardware clock (Appendix C of this AMX Tool Guide). If necessary, edit the driver to meet your requirements or create your own clock driver as described in Chapter 5.2 of the AMX User's Guide. Assemble or compile the driver.

2. Use the AMX Configuration Manager to create a User Parameter File describing your AMX application requirements (Chapter 14 of the AMX User's Guide). If necessary, edit the User Parameter File to adapt the configuration extensions identified by the keyword `...EXT` to meet your toolset requirements as defined in the toolset specific chapter of this guide. Then use the Manager to generate your System Configuration Module. Assemble the module.

   If you are using the PC Supervisor, use the AMX Configuration Manager to describe your PC Supervisor application requirements (Chapter 2.2 of the AMX 86 PC Supervisor Reference Manual). Be sure to select the AMX PC Supervisor clock task and adjust the AMX clock frequency for 18, 20 or 100 Hz operation.

3. If you are NOT placing AMX into its own private ROM, separate from your application, go to step 7.

4. Copy the AMX ROM Option Definitions file `AMX831RO.DEF` and edit the copy to select the AMX Managers which are to be in the ROM and to locate the ROM (Chapter 15.8 of the AMX User's Guide).

5. Using the modified ROM Option Definitions file from step 4, assemble the AMX ROM Option Module `AA831ROP.ASM` and link it with the AMX Library to create your AMX ROM (Chapter 15.8 of the AMX User's Guide).

6. Using the modified ROM Option Definitions file from step 4, assemble the AMX ROM Access Module `AA831RAC.ASM` (Chapter 15.8 of the AMX User's Guide).

7. Compile your `main()` C program and AMX application modules. You can use the AMX Sample Program `AM831SAM.C` (described in Chapter 4 of the AMX C Programming Guide) as a model and add enhancements if warranted. You may wish to create custom error handling procedures as described in Chapters 13.1 and 13.2 of the AMX User's Guide.

8. Link the modules from steps 1, 2, 6 (if step done) and 7 with the AMX Library and the C Library to create your AMX application load module. Follow the linking instructions provided in this guide.

2. Paradigm (PD and PX) Tool Guide

AMX™ 86 has been developed on a PC with Windows® NT v4.0 using the Paradigm Systems Inc. software development tools listed below. The AMX libraries and object modules on the product disks have been generated using the most recent tools listed. If you are not using this toolset, you may have to rebuild the AMX libraries in order to use your out-of-date tools.

Toolset *PD* uses the Paradigm tools to generate AMX applications for conventional 80x86 processors which employ 16-bit (1 Mb) memory addressing.

Toolset *PX* uses the same tools to generate AMX applications for VAutomation processors which allow 24-bit (16 Mb) memory addressing.

		v6.0
PASM	80x86 Assembler (Note 1)	v5.0
PCC	C/C++ compiler	v6.0
PLIB	Librarian	v6.0
PLINK	Linker	v6.0
LOCATE	Locator	v6.0
	IDE Debugger	

Note 1: The assembler provided with Paradigm v5.0 and v6.0 tools identifies itself as v5.0.

AMX 86 has been tested on the following platforms.

80x86 PC treated as an embedded processor
AMD Net186™ Demonstration Board
VAutomation *i*CON186™ TIPS3 Evaluation Board

IDE or Command Line

The Paradigm C/C++ Integrated Development Environment (IDE) provides a Windows® environment in which the Paradigm tools can be used to create and test your AMX application. The tools can also be invoked directly from the Windows command prompt. This tool guide describes the use of the command line tools, identifying the particular command line switches required for construction of your AMX application.

Most developers will prefer to use the Paradigm IDE in which command line switches are replaced by IDE option settings. Browse the HTML manual *AMX86_PD.HTM* in installation directory *AMX831\MANUALS\PARADIGM*. It describes how the Paradigm project *AM831SAM.IDE* in directory *AMX831\TOOLPD\IDE* was created and then used to build the AMX 86 Sample Program.

> Warning
>
> Do not mix toolset *PD* files (installation directory *TOOLPD*) with toolset *PX* files (installation directory *TOOLPX*).

Environment Variables

Set the following DOS environment variables to provide access to all AMX and Paradigm tools, header files, object files and libraries.

AMXPATH	Path to AMX installation directory (*...\AMX831*)
PATH	Path to AMX and Paradigm executable programs
INCLUDE	Path to all Paradigm include header files
LIB	Path to all Paradigm object files and libraries
TMP	Path to a temporary directory for use by tools

Register Usage

The Paradigm version of AMX makes the following C interface register assumptions. Registers *AX*, *CX* and *DX* can always be altered by C procedures. Registers *BX*, *SI*, *DI*, *BP*, *SP* and all segment registers are preserved by AMX according to the Paradigm rules for C procedures. Integers are returned from C procedures in register *AX*. Longs and pointers are returned from C procedures in register pair *DX:AX*. The *DS* register is dedicated for access to global data in segment *DGROUP*. You must NOT use any C compilation switch which changes these register assumptions.

Stack Checking

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

This stack checking is a compilation option which is normally disabled. Although it can be enabled with the *-N* switch during compilation (see Paradigm *PCC* switch options), its use must be avoided.

If you cannot avoid the use of modules compiled with stack checking enabled, you may be able to defeat the run-time stack checking by judiciously setting the value of global variable *_stklen* as described for the Borland tools (see Chapter 3).

Making Libraries

To make a library from a collection of object modules, create a library specification file *YOURLIB.LBM*. Use the Paradigm version of the AMX library specification file *AMX831.LBM* as a guide. Make your library as follows.

```
PLIB @YOURLIB.LBM
```

Assembling the AMX System Configuration Module

Your AMX System Configuration Module *SYSCFG.ASM* is assembled using Paradigm's *PASM* assembler as follows. All AMX header files *AMX831xx.DEF* must be present in the current directory together with file *SYSCFG.ASM*.

```
PASM /ML /N SYSCFG
```

Using the Paradigm C/C++ Compiler

All AMX header files *AMX831xx.H* must be present in the current directory together with your source file being compiled.

By default, the Paradigm compiler passes function parameters on the stack. It adds leading underscores to both function names and variable names. These conventions are compatible with the AMX C interface. Paradigm also supports the *cdecl* keyword which defines the C parameter passing convention with which AMX is compatible. The AMX keyword *AMCCPP* is defined to be *cdecl* forcing all AMX procedures and AMX-callable application procedures to pass parameters on the stack.

It is important to note that all application functions which are called by AMX must also be declared with the *cdecl* keyword. All tasks, Restart and Exit Procedures, Timer Procedures and Task Termination Procedures must be declared with the *cdecl* keyword. So must your User Error Procedure, Fatal Exit Procedure, Time/Date Scheduling Procedure and Memory Assignment Procedure.

By default, the AMX C header file *AMX831CF.H* assumes that the Microsoft compiler is being used. Therefore, you must define symbol *AMCCIF=2* on the compiler command line to force the AMX header files to select the characteristics of the Borland compiler from which the Paradigm compiler was derived.

Use the following compilation switches when you are compiling modules for use in the AMX environment.

```
by default          ; pass all parameters on stack and
                    ; add leading underscores to names
                    ; of procedures declared cdecl
by default          ; add leading underscores to variable names
-oFILENAME.OBJ      ; output object module FILENAME.OBJ
-DAMCCIF=2          ; AMX headers select Paradigm
-ml                 ; use Intel Large model but peg DS to DGROUP
-c                  ; compile but do not link
-1                  ; use 80186 instructions (for Turbo86 or Turbo186)
-Y                  ; use 24-bit addressing (mandatory for toolset PX only)
-O                  ; (optional) enable optimization
-f-                 ; (optional) no floating point used
-v                  ; (optional) generate debug information for
                    ; Turbo Debugger
```

The compilation command line for 16-bit toolset *PD* is therefore of the form:

```
PCC -ml -c -1 -O -DAMCCIF=2 FILENAME.C
```

The compilation command line for 24-bit toolset *PX* is therefore of the form:

```
PCC -ml -c -1 -O -Y -DAMCCIF=2 FILENAME.C
```

Linking with the Paradigm Linker

The modules which form your AMX 86 system must be linked in the following order.

```
C0L.OBJ or          ; Paradigm C 16-bit startup module
C0LX.OBJ            ; Paradigm C 24-bit startup module
SYSCFG.OBJ          ; AMX System Configuration Module

Your MAIN module
Other application modules

CH8253T.OBJ         ; AMX 8253 clock driver or your equivalent
or
CH186EST.OBJ        ; AMX Am186ES clock driver or your equivalent


The following KwikLook object modules (only if KwikLook Manager used)
BJ840STB.OBJ        ; KwikLook Stub module
BJ840DRV.OBJ        ; KwikLook Device Driver
CHnnnnnS.OBJ        ; AMX chip-specific serial driver or your equivalent


AA831BKE.OBJ        ; AMX Breakpoint exclusion module
                    ; (see Chapter 13.6 in AMX 86 User's Guide)

AJ831CV.OBJ         ; AMX 86 v2 to v3 conversion module
                    ; (only if converting an AMX 86 v2 application)

AA831RAC.OBJ        ; AMX ROM Access Module (customized)
                    ; (only if AMX placed in a separate ROM)
                    ; (see Appendix E of this manual)

AMX831.LIB          ; AMX 86 Library

Paradigm C Runtime Libraries for target hardware
```

> **Note**
>
> Because the Paradigm tools do not support the development of DOS applications, the AMX 86 PC Supervisor cannot be used with Paradigm toolset *PD* or *PX* .

Create a link specification file *YOURLINK.LKS*. Use the Paradigm version of the AMX 86 Sample Program link specification file *AM831SAM.LKS* as a guide. If you are using the *KwikLook* Manager, use the Paradigm version of the *KwikLook* Sample Program link specification file *BJSAMPLE.LKS* as a guide.

Create a locate specification (configuration) file *YOURLINK.CFG*. Use the Paradigm version of the AMX 86 Sample Program locate specification file *AM831SAM.CFG* as a guide.

> Note
>
> If you decide to omit any of the link commands which are in the sample link/locate specifications, you may encounter link errors or run-time faults.

Link with the Paradigm linker using the following linker options on the command line or in the link specification file.

```
/m          ; add public symbols to the map file
/n          ; no default libraries
/k          ; suppress no stack warning (used for ROM Option)
/Tee        ; use 24-bit addressing (mandatory for toolset PX only)
/Sd:200     ; default stack size
/Hd:200     ; default heap size
/v          ; add debug information for IDE debugger use
```

The link command line for 16-bit toolset *PD* is therefore of the form:

```
PLINK /m /n /Sd:200 /Hd:200 @YOURLINK.LKS
```

The link command line for 24-bit toolset *PX* is therefore of the form:

```
PLINK /m /n /Tee /Sd:200 /Hd:200 @YOURLINK.LKS
```

The resulting load module *YOURLINK.EXE* must be converted to a form suitable for use with the Paradigm IDE debugger. Locate with the Paradigm *LOCATE* tool using the following command line options.

```
-LnYOURLINK.RPT   ; list file is named YOURLINK.RPT
-Lp               ; add public symbols to list file
-Lr               ; add region map to list file
-Ls               ; add segment map to list file
YOURLINK          ; use configuration file YOURLINK.CFG
                  ; generate output file YOURLINK.AXE
```

The locate command line for toolset *PD* or *PX* is therefore of the form:

```
LOCATE -LnYOURLINK.RPT -Lp -Lr -Ls YOURLINK
```

16-Bit and 24-Bit Operation

For 16-bit (1 Mb) applications, Paradigm toolset *PD* must be used. Toolset dependent files will be found in installation directory *AMX831\TOOLPD*. All memory segments and normalized pointers are paragraph (16 byte) aligned.

For 24-bit (16 Mb) applications, Paradigm toolset *PX* must be used. Toolset dependent files will be found in installation directory *AMX831\TOOLPX*. All memory segments and normalized pointers are page (256 byte) aligned.

The AMX PC Supervisor is not supported for either 16-bit or 24-bit operation. All other AMX features are available for use with either toolset.

Memory Manager Caveats

The AMX 86 Memory Manager provides fast and efficient allocation of paragraph (16-byte) aligned blocks of memory for 80x86 systems with 16-bit memory addressing.

When used with the VAutomation Turbo86 processors with 24-bit memory addressing, memory blocks are still allocated with 16-byte alignment. Since the blocks are not page (256 byte) aligned, the pointer to an allocated memory block will not be a normalized 24-bit pointer. If the allocated block is no more than *0xFF00* bytes in size, then all of the bytes in the block will be directly accessible using the pointer provided by the Memory Manager. If not, you must use a normalized pointer to access any memory location which is at least *0xFF00* bytes higher in memory than the base of the memory block.

If the linker declares that **symbol *AM24ERR* is undefined**, then you have attempted to link a 16-bit AMX 86 configuration which uses the AMX Memory Manager with the 24-bit AMX 86 Library *AMX831.LIB* from toolset directory *TOOLPX\LIB*. Use the AMX Configuration Builder to edit your User Parameter File to enable 24-bit addressing in the System Parameter window or link with the 16-bit library from director *TOOLPD\LIB*.

If the linker declares that **symbol *AM20ERR* is undefined**, then you have attempted to link a 24-bit AMX 86 configuration with the 16-bit AMX 86 Library *AMX831.LIB* from toolset directory *TOOLPD\LIB*. Use the AMX Configuration Builder to edit your User Parameter File to disable 24-bit addressing in the System Parameter window or link with the 24-bit library from director *TOOLPX\LIB*.

If you are using the AMX ROM Option to locate AMX in a separate ROM (see Appendix F), the linker cannot detect a 16/24-bit Memory Manager mismatch between your AMX ROM and your AMX System Configuration Module. If your configuration does not match the AMX ROM, the Memory Manager will force an AMX fatal exit with fatal exit code *AERFX6*.

Linking a Separate AMX ROM

AMX 86 can be committed to a separate ROM as described in Appendix E of this manual. Copy the following files to the current directory.

AA831ROS.OBJ	AMX ROM Option Entry Module
AMX831 .LIB	AMX Library
AA831K .DEF	AMX Private Definitions
AMX831EC.DEF	AMX Error Code Definitions
AMX831RO.DEF	AMX ROM Option Definitions
AMX831RO.CFG	AMX ROM Option Locate Specification File
AA831ROP.ASM	AMX ROM Option Module
AA831RAC.ASM	AMX ROM Access Module

Edit the copy of the AMX ROM Option Definitions file *AMX831RO.DEF* to define your ROM option specifications. You must also edit the copy of the AMX ROM Option Locate Specification File *AMX831RO.CFG* to specify the location of your AMX ROM and the AMX Data Segment.

Assemble the ROM Option and ROM Access Modules as follows.

```
PASM /ML /N AA831ROP

PASM /ML /N AA831RAC
```

When you link your AMX application, be sure to include your customized AMX ROM Access Module *AA831RAC.OBJ* (created above) in your system link specification file.

The AMX ROM is linked and located with the Paradigm linker and locator for 16-bit toolset *PD* as follows.

```
PLINK /m /n /k AA831ROP+AA831ROS,AMX831RO,AMX831RO,AMX831.LIB
LOCATE -LnAMX831RO.RPT -Lp -Lr -Ls AMX831RO
```

The AMX ROM is linked and located with the Paradigm linker and locator for 24-bit toolset *PX* as follows.

```
PLINK /m /n /k /Tee AA831ROP+AA831ROS,AMX831RO,AMX831RO,AMX831.LIB
LOCATE -LnAMX831RO.RPT -Lp -Lr -Ls AMX831RO
```

This example generates file *AMX831RO.AXE* in the file format supported by the Paradigm IDE debugger. The Paradigm tools can be used to convert this absolute file to an image format suitable for transfer to ROM.

Paradigm IDE Debugger

The Paradigm IDE debugger is a Windows® based debugger which supports source level debugging of your AMX 86 system.

The Paradigm debugger can operate using an in-circuit emulator connected to your 80x86 target hardware. Check with Paradigm to determine which emulators are supported. The debugger can also be used with the FS2 VSA-186 VAutomation System Analyzer and its JTAG connection to the VAutomation *i*CON186 TIPS3 Evaluation Board.

When used with hardware assisted breakpointing, your target processor is effectively halted while at breakpoints. Hence there is no need to incorporate the AMX Breakpoint Manager into your system.

The Paradigm IDE debugger can also operate using a serial (or other) connection to the system under test. When used in this fashion, you must install the Paradigm remote debug monitor in your target hardware. Instructions for doing so are provided by Paradigm. Your version of the target monitor must provide a device driver for the serial (or other) device used for communication with the host debugger. It is recommended that your driver use polled I/O so that the target monitor can operate with interrupts disabled.

When using the Paradigm target monitor, you may find it advantageous to use the AMX Breakpoint Manager to ensure that all task activity halts when you encounter a breakpoint. To include the Breakpoint Manager in your configuration using the AMX Configuration Manager, go to the Breakpoint Parameter window and check the box to enable the Breakpoint Manager. Set the breakpoint entry delay to 2 and the breakpoint exit delay to 20.

Using the *KwikLook* Fault Finder

The *KwikLook*™ Fault Finder is compatible with the Paradigm IDE debugger providing full screen, source level, task-aware debugging from within the Microsoft Windows environment. *KwikLook* can be invoked directly from the debugger while at breakpoints giving you finger tip access to your application from the AMX perspective. Note that *KwikLook* and the Paradigm debugger share a common link to the target system.

AMX 86 Configuration Extensions

There should be no need to use the AMX Breakpoint Manager if you are using the Paradigm debugger with the FS2 VSA-186 VAutomation System Analyzer or an in-circuit emulator. The Breakpoint Manager can be of assistance if you are using the Paradigm remote monitor to connect to the host debugger.

If you use the AMX Breakpoint Manager, your AMX System Configuration Module will include the necessary breakpoint delay parameters. However, you may wish to revise the *...EXT* extension directive in your AMX User Parameter File to add the features described below.

The breakpoint delays can be edited using the AMX Configuration Builder. In rare cases, you may have to alter extension parameters that are not handled by the Configuration Builder. To do so, find and edit the following statement in your User Parameter File *SYSCFG.UP*.

```
...EXT 0,0,DBGA,BPED,BPXD,PCSOPT
```

where:

0,0 = Reserved

DBGA = 0 = Debugger attributes
Bit 0 = 0 = reserved
Bit 1 = 1 if using a debugger which, at breakpoints, takes over the DOS INT 21H software interrupt and does not restore it upon continuation.
Bits 2 to 31 = 0 = reserved

BPED = n = Breakpoint entry delay (0 < *n* < 32768 AMX ticks)
Use *n = 2* for Paradigm.

BPXD = n = Breakpoint exit delay (0 < *n* < 32768 AMX ticks)
Use *n = 20* for Paradigm.

PCSOPT = 0 = PC Supervisor extension options (not supported for this toolset)

For example, when using the Paradigm debugger with the remote monitor, your User Parameter File *SYSCFG.UP* to should contain the following statement.

```
...EXT 0,0,0,2,20,0
```

If you edit the *...EXT* directive, use the Configuration Builder to generate a new copy of your System Configuration Module *SYSCFG.ASM*. Be sure to use the most recently issued version of the AMX Configuration Template File *AM831CG.CT*.

3. Borland (TC) Tool Guide

AMX™ 86 has been developed on a PC with Windows® NT v4.0 using the Borland software development tools listed below. The AMX libraries and object modules on the product disks have been generated using the most recent tools listed. If you are not using this toolset, you may have to rebuild the AMX libraries in order to use your out-of-date tools.

		v2.0	v4.02	v4.52	v5.0
TASM	80x86 Assembler (Note 1)	v1.0	v3.1	v4.1	v4.1
TCC	Turbo C compiler (Note 2)	v2.0			
BCC	Turbo C/C++ compiler		v4.02	v4.52	v5.0
TLIB	Librarian	v2.0	v4.00	v4.00	v4.00
TLINK	Linker	v2.0	v6.10	v7.00	v7.1
TD	Turbo Debugger	v1.0	v4.02	v4.6	v5.0

Note 1: The assembler provided with Borland Turbo Assembler v4.0 and v5.0 identifies itself as v4.1.

Note 2: With the release of Turbo C++ v2.0, Borland changed the compiler name from *TCC.EXE* to *BCC.EXE*.

AMX 86 has been tested on the following platforms.

80x86 PC with MS-DOS v3 to MS-DOS v6.22
AMD Net186™ Demonstration Board
VAutomation *i*CON186 TIPS3 Evaluation Board

Environment Variables

Set the following DOS environment variables to provide access to all AMX and Borland tools, header files, object files and libraries.

AMXPATH	Path to AMX installation directory (*...\AMX831*)
PATH	Path to AMX and Borland executable programs
INCLUDE	Path to all Borland include header files
LIB	Path to all Borland object files and libraries
TMP	Path to a temporary directory for use by tools

Register Usage

The Borland version of AMX makes the following C interface register assumptions. Registers *AX*, *CX* and *DX* can always be altered by C procedures. Registers *BX*, *SI*, *DI*, *BP*, *SP* and all segment registers are preserved by AMX according to the Borland rules for C procedures. Integers are returned from C procedures in register *AX*. Longs and pointers are returned from C procedures in register pair *DX:AX*. The *DS* register is dedicated for access to global data in segment *DGROUP*. You must NOT use any C compilation switch which changes these register assumptions.

Stack Checking

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

This stack checking is a compilation option which is normally disabled. Although it can be enabled with the *-N* switch during compilation (see Borland *BCC* switch options), its use must be avoided.

You may be using some C modules which were compiled with stack checking enabled. In order to defeat Borland's runtime stack checking, you must set global unsigned integer variable *_stklen* to reflect the largest stack required by any of your AMX tasks or by your *main* program function. To disable stack checking completely, set *_stklen* to *0xFF00*. The initialization of *_stklen* should be done as the first step in your *main* program function. Note that the variable may appear in link maps as *__stklen* (double underscore) since a leading underscore is added to variable names by the C compiler.

```
extern unsigned int _stklen;
_stklen = 0xFF00;
```

> Warning!
>
> Do not forget to set *_stklen* to *0xFF00* in your *main* program to inhibit run time stack checking by functions in Borland's library.

Making Libraries

To make a library from a collection of object modules, create a library specification file *YOURLIB.LBM*. Use the Borland version of the AMX library specification file *AMX831.LBM* as a guide. Make your library as follows.

```
TLIB @YOURLIB.LBM
```

Assembling the AMX System Configuration Module

Your AMX System Configuration Module *SYSCFG.ASM* is assembled using Borland's *TASM* assembler as follows. All AMX header files *AMX831xx.DEF* must be present in the current directory together with file *SYSCFG.ASM*.

```
TASM /ML /N SYSCFG
```

Using the Borland C/C++ Compiler

All AMX header files *AMX831xx.H* must be present in the current directory together with your source file being compiled.

By default, the Borland compiler passes function parameters on the stack. It adds leading underscores to both function names and variable names. These conventions are compatible with the AMX C interface. Borland also supports the *cdecl* keyword which defines the C parameter passing convention with which AMX is compatible. The AMX keyword *AMCCPP* is defined to be *cdecl* forcing all AMX procedures and AMX-callable application procedures to pass parameters on the stack.

It is important to note that all application functions which are called by AMX must also be declared with the *cdecl* keyword. All tasks, Restart and Exit Procedures, Timer Procedures and Task Termination Procedures must be declared with the *cdecl* keyword. So must your User Error Procedure, Fatal Exit Procedure, Time/Date Scheduling Procedure and Memory Assignment Procedure.

By default, the AMX C header file *AMX831CF.H* assumes that the Microsoft compiler is being used. Therefore, you must define symbol *AMCCIF=2* on the compiler command line to force the AMX header files to select the Borland compiler characteristics.

Use the following compilation switches when you are compiling modules for use in the AMX environment.

```
by default          ; pass all parameters on stack and
                   ; add leading underscores to names
                   ; of procedures declared cdecl
by default         ; add leading underscores to variable names
by default         ; output object module FILENAME.OBJ
-DAMCCIF=2         ; AMX headers select Borland
-ml                ; use Intel Large model but peg DS to DGROUP
-c                 ; compile but do not link
-O                 ; (optional) enable optimization
-f-                ; (optional) no floating point used
-v                 ; (optional) generate debug information for
                   ; Turbo Debugger
```

The compilation command line is therefore of the form:

```
BCC -ml -c -DAMCCIF=2 FILENAME.C
```

Linking with the Borland Linker

The modules which form your AMX 86 system must be linked in the following order.

```
AJ831PRF.OBJ        ; AMX Segment Prefix Module
C0L.OBJ             ; Borland C startup module
SYSCFG.OBJ          ; AMX System Configuration Module

Your MAIN module
Other application modules

AMX PC/AT or equivalent clock driver:
CH8253T.OBJ         ; AMX 8253 clock driver or your equivalent
or
CH_PC_T.OBJ         ; AMX PC BIOS clock driver


The following KwikLook object modules (only if KwikLook Manager used)
BJ840STB.OBJ        ; KwikLook Stub module
BJ840DRV.OBJ        ; KwikLook Device Driver
CHnnnnnS.OBJ        ; AMX chip-specific serial driver or your equivalent


AA832LTC.OBJ        ; (only if creating a DOS resident system)
                    ; (see Chapter 6.1 in PC Supervisor Reference Manual)

AA831BKE.OBJ        ; AMX Breakpoint exclusion module
                    ; (see Chapter 13.6 in AMX 86 User's Guide)

AJ831CV.OBJ         ; AMX 86 v2 to v3 conversion module
                    ; (only if converting an AMX 86 v2 application)

AA831RAC.OBJ        ; AMX ROM Access Module (customized)
                    ; (only if AMX placed in a separate ROM)
                    ; (see Appendix E of this manual)

AMX832.LIB          ; AMX 86 PC Supervisor Library
                    ; (only if PC Supervisor or its drivers are used)

AMX831.LIB          ; AMX 86 Library

Borland C Runtime Libraries for target hardware
```

Create a link specification file *YOURLINK.LKS*. Use the Borland version of the AMX 86 Sample Program link specification file *AM831SAM.LKS* or the AMX PC Supervisor Test Program link specification file *AMX832T.LKS* as a guide. If you are using the *KwikLook* Manager, use the Borland version of the *KwikLook* Sample Program link specification file *BJSAMPLE.LKS* as a guide.

> Note
>
> If you decide to omit any of the link commands which are in the sample link specifications, you may encounter link errors or run-time faults.

Link with the Borland linker using the following linker options on the command line or in the link specification file.

```
/m          ; add public symbols to the map file
/n          ; no default libraries
/v          ; add debug information for Turbo Debugger use
```

The link command line is therefore of the form:

```
TLINK @YOURLINK.LKS
```

The resulting load module *YOURLINK.EXE* is suitable for use with the Borland Turbo Debugger.

Linking a Separate AMX ROM

AMX 86 can be committed to a separate ROM as described in Appendix E of this manual. Copy the following files to the current directory.

AA831ROS.OBJ	AMX ROM Option Entry Module
AMX831 .LIB	AMX Library
AA831K .DEF	AMX Private Definitions
AMX831EC.DEF	AMX Error Code Definitions
AMX831RO.DEF	AMX ROM Option Definitions
AA831ROP.ASM	AMX ROM Option Module
AA831RAC.ASM	AMX ROM Access Module

Edit the copy of the AMX ROM Option Definitions file *AMX831RO.DEF* to define your ROM option specifications.

Assemble the ROM Option and ROM Access Modules as follows.

```
TASM /ML /N AA831ROP

TASM /ML /N AA831RAC
```

When you link your AMX application, be sure to include your customized AMX ROM Access Module *AA831RAC.OBJ* (created above) in your system link specification file.

The AMX ROM is then linked with the Borland linker as follows.

```
TLINK AA831ROP+AA831ROS,AMXROM /m /n,AMXROM,AMX831.LIB
```

This example generates file *AMXROM.EXE* in DOS executable file format. This execute file must be converted to a ROM image. The manner in which this is accomplished will depend upon the development tools which you are using for that purpose.

Borland Turbo Debugger

The Borland DOS and Windows® based Turbo Debugger supports source level debugging of your AMX 86 system.

Use the AMX Breakpoint Manager to ensure that all task activity halts when you encounter a breakpoint. To include the Breakpoint Manager in your configuration using the AMX Configuration Manager, go to the Breakpoint Parameter window and check the box to enable the Breakpoint Manager. Set the breakpoint entry delay to 2 and the exit delay to 20.

To use the Borland Turbo Debugger in its remote mode, you must install the Borland remote debugging driver in your target hardware. Instructions for doing so are provided in the Turbo Debugger Reference Manual.

If you use the AMX 86 PC Supervisor, be sure to include the PC Supervisor Debug Module so that both you and the debugger can share DOS and the PC devices. To include the PCS Debug Module in your configuration using the AMX Configuration Manager, go to the PC Supervisor Parameter window and check the box labeled "Debug Module required".

You should be aware that the debugger may make DOS calls while it traces your code. Because of this, you may encounter interference with your application's use of DOS and the PC devices.

InSight Discontinued

The DOS-based InSight™ Debug Tool is no longer supported. It has been superceded by the AMX 86 *KwikLook* Fault Finder.

Existing InSight users can continue to use InSight with AMX 86. You must follow the directions for linking and debugging provided in earlier releases of the AMX 86 Tool Guide and in the InSight Reference Manual.

To use the updated AMX Configuration Builder for Windows with InSight, go to the Configuration Manager's Edit menu and select Preferences... from the menu. Check the box labeled "Show InSight configuration page". Then exit from the Configuration Manager and restart the manager. The InSight Parameter window will then be accessible via the InSight tab. There has been no change to the use and meaning of the InSight parameters presented in that window.

AMX 86 Configuration Extensions

If you use the AMX Breakpoint Manager, your AMX System Configuration Module will include the necessary breakpoint delay parameters. However, you may wish to revise the *...EXT* extension directive in your AMX User Parameter File to add the features described below. The extension is also used to provide additional PC Supervisor options.

The breakpoint delays and PC Supervisor options can be edited using the AMX Configuration Builder. In rare cases, you may have to alter extension parameters that are not handled by the Configuration Builder. To do so, find and edit the following statement in your User Parameter File *SYSCFG.UP*.

```
...EXT 0,0,DBGA,BPED,BPXD,PCSOPT
```

where:

0,0 = Reserved

DBGA = 0 = Debugger attributes
Bit 0 = 0 = reserved
Bit 1 = 1 if using a debugger which, at breakpoints, takes over the DOS INT 21H software interrupt and does not restore it upon continuation.
Bits 2 to 31 = 0 = reserved

BPED = n = Breakpoint entry delay (0 < *n* < 32768 AMX ticks)
Use *n = 2* for Borland.

BPXD = n = Breakpoint exit delay (0 < *n* < 32768 AMX ticks)
Use *n = 20* for Borland.

PCSOPT = PC Supervisor extension options
Bit 0 = 1 if the PC Supervisor driver for the Parallel Port Adapters should pass on all *INT 17H* requests for I/O on undefined or unused *LPTn* devices to the original BIOS handler.

You can use this option to force the PPA driver v3.02 or later to behave in the same manner as earlier versions.
Bits 1 to 31 = 0 = reserved

For example, when using the Borland tools, your User Parameter File *SYSCFG.UP* to should contain the following statement.

```
...EXT 0,0,0,2,20,0
```

If you edit the *...EXT* directive, use the Configuration Builder to generate a new copy of your System Configuration Module *SYSCFG.ASM*. Be sure to use the most recently issued version of the AMX Configuration Template File *AM831CG.CT*.

4. Microsoft (MC) Tool Guide

AMX™ 86 has been developed on a PC with Windows® NT v4.0 using the Microsoft software development tools listed below. The AMX libraries and object modules on the product disks have been generated using the most recent tools listed. If you are not using this toolset, you may have to rebuild the AMX libraries in order to use your out-of-date tools.

		v6.0	v7.0	vc1.5	vc1.52
MASM	80x86 Assembler	v6.00	v6.00	v6.11	v6.11d
CL	C/C++ compiler (Note 1)	v6.0	v7.0	v8.00c	v8.00c
LIB	Librarian	v3.18	v3.20	v3.40	v3.40
LINK	Linker	v5.13	v5.30	v5.60	v5.60
CV	CodeView Debugger	v3.14	v4.0	v4.10	v4.10

Note 1: The compiler provided with Microsoft Visual C/C++ v1.5 and v1.52 identifies itself as v8.00c.

AMX 86 has been tested on the following platforms.

80x86 PC with MS-DOS v3 to MS-DOS v6.22
AMD Net186™ Demonstration Board
VAutomation *i*CON186™ TIPS3 Evaluation Board

Environment Variables

Set the following DOS environment variables to provide access to all AMX and Microsoft tools, header files, object files and libraries.

AMXPATH	Path to AMX installation directory (*...\AMX831*)
PATH	Path to AMX and Microsoft executable programs
INCLUDE	Path to all Microsoft include header files
LIB	Path to all Microsoft object files and libraries
TMP	Path to a temporary directory for use by tools

Register Usage

The Microsoft version of AMX makes the following C interface register assumptions. Registers *AX*, *CX* and *DX* can always be altered by C procedures. Registers *BX*, *SI*, *DI*, *BP*, *SP* and all segment registers are preserved by AMX according to the Microsoft rules for C procedures. Integers are returned from C procedures in register *AX*. Longs and pointers are returned from C procedures in register pair *DX:AX*. The *DS* register is dedicated for access to global data in segment *DGROUP*. You must NOT use any C compilation switch which changes these register assumptions.

Stack Checking

The Microsoft C Compiler generates a runtime stack check at the entry point to every C function. The check verifies that, after local variable storage has been allocated on the stack, stack still remains available for use.

This stack checking is a compilation option which can be inhibited with the *-Gs* switch during compilation (see Microsoft *CL* switch options). 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 functions, you will have stack checking in effect.

In order to defeat Microsoft's runtime stack checking, KADAK provides an alternate stack check procedure in object module *AJ831MC.OBJ*. This module must be linked into your system prior to linking the Microsoft C Runtime Library. Stack checking is completely inhibited by this new module.

> Warning!
>
> Do not forget to link object module *AJ831MC.OBJ* with your application to inhibit run time stack checking by functions in Microsoft's library.

Making Libraries

To make a library from a collection of object modules, create a library specification file *YOURLIB.LBM*. Use the Microsoft version of the AMX library specification file *AMX831.LBM* as a guide. Make your library as follows.

```
LIB @YOURLIB.LBM
```

Assembling the AMX System Configuration Module

Your AMX System Configuration Module *SYSCFG.ASM* is assembled using Microsoft's *MASM* assembler as follows. All AMX header files *AMX831xx.DEF* must be present in the current directory together with file *SYSCFG.ASM*.

```
MASM SYSCFG /ML /N,,;
```

Using the Microsoft C/C++ Compiler

All AMX header files *AMX831xx.H* must be present in the current directory together with your source file being compiled.

By default, the Microsoft compiler passes function parameters on the stack. It adds leading underscores to both function names and variable names. These conventions are compatible with the AMX C interface. Microsoft also supports the *cdecl* keyword which defines the C parameter passing convention with which AMX is compatible. The AMX keyword *AMCCPP* is defined to be *cdecl* forcing all AMX procedures and AMX-callable application procedures to pass parameters on the stack.

It is important to note that all application functions which are called by AMX must also be declared with the *cdecl* keyword. All tasks, Restart and Exit Procedures, Timer Procedures and Task Termination Procedures must be declared with the *cdecl* keyword. So must your User Error Procedure, Fatal Exit Procedure, Time/Date Scheduling Procedure and Memory Assignment Procedure.

By default, the AMX C header file *AMX831CF.H* assumes that the Microsoft compiler is being used. You can, if you wish, define symbol *AMCCIF=1* on the compiler command line to force the AMX header files to select the Microsoft compiler characteristics.

Use the following compilation switches when you are compiling modules for use in the AMX environment.

```
by default          ; pass all parameters on stack and
                    ; add leading underscores to names
                    ; of procedures declared cdecl
by default          ; add leading underscores to variable names
by default          ; output object module FILENAME.OBJ
/DAMCCIF=1          ; (optional) by default AMX headers select Microsoft
/Alfw               ; use Intel Large model but peg DS to DGROUP
/Gs                 ; no stack checking
/Zp                 ; pack structures
/Ze                 ; enable language extensions
/c                  ; compile but do not link
/f-                 ; (optional) use optimizing compiler
/Zi or /Z7          ; (optional) generate debug information for
                    ; CodeView Debugger
```

The compilation command line is therefore of the form:

```
CL /Alfw /Gs /Zp /Ze /c FILENAME.C
```

Linking with the Microsoft Linker

The modules which form your AMX 86 system must be linked in the following order.

```
AJ831PRF.OBJ        ; AMX Segment Prefix Module
SYSCFG.OBJ          ; AMX System Configuration Module

Your MAIN module
Other application modules

AMX PC/AT or equivalent clock driver:
CH8253T.OBJ         ; AMX 8253 clock driver or your equivalent
or
CH_PC_T.OBJ         ; AMX PC BIOS clock driver


The following KwikLook object modules (only if KwikLook Manager used)
BJ840STB.OBJ        ; KwikLook Stub module
BJ840DRV.OBJ        ; KwikLook Device Driver
CHnnnnnS.OBJ        ; AMX chip-specific serial driver or your equivalent


AJ831MC.OBJ         ; AMX Microsoft Interface
AA832LMC.OBJ        ; (only if creating a DOS resident system)
                    ; (see Chapter 6.1 in PC Supervisor Reference Manual)

AA831BKE.OBJ        ; AMX Breakpoint exclusion module
                    ; (see Chapter 13.6 in AMX 86 User's Guide)

AJ831CV.OBJ         ; AMX 86 v2 to v3 conversion module
                    ; (only if converting an AMX 86 v2 application)

AA831RAC.OBJ        ; AMX ROM Access Module (customized)
                    ; (only if AMX placed in a separate ROM)
                    ; (see Appendix E of this manual)

AMX832.LIB          ; AMX 86 PC Supervisor Library
                    ; (only if PC Supervisor or its drivers are used)

AMX831.LIB          ; AMX 86 Library

Microsoft C Runtime Libraries for target hardware
```

Create a link specification file *YOURLINK.LKS*. Use the Microsoft version of the AMX 86 Sample Program link specification file *AM831SAM.LKS* or the AMX PC Supervisor Test Program link specification file *AMX832T.LKS* as a guide. If you are using the *KwikLook* Manager, use the Microsoft version of the *KwikLook* Sample Program link specification file *BJSAMPLE.LKS* as a guide.

> Note
>
> If you decide to omit any of the link commands which are in the sample link specifications, you may encounter link errors or run-time faults.

Link with the Microsoft linker using the following linker options on the command line or in the link specification file.

```
/MAP        ; add public symbols to the map file
/NOE        ; no extended dictionary search for library references
/CO         ; add debug information for CodeView use
```

The link command line is therefore of the form:

```
LINK @YOURLINK.LKS
```

The resulting load module *YOURLINK.EXE* is suitable for use with the Microsoft CodeView Debugger.

Linking a Separate AMX ROM

AMX 86 can be committed to a separate ROM as described in Appendix E of this manual. Copy the following files to the current directory.

AA831ROS.OBJ	AMX ROM Option Entry Module
AMX831 .LIB	AMX Library
AA831K .DEF	AMX Private Definitions
AMX831EC.DEF	AMX Error Code Definitions
AMX831RO.DEF	AMX ROM Option Definitions
AA831ROP.ASM	AMX ROM Option Module
AA831RAC.ASM	AMX ROM Access Module

Edit the copy of the AMX ROM Option Definitions file *AMX831RO.DEF* to define your ROM option specifications.

Assemble the ROM Option and ROM Access Modules as follows.

```
MASM AA831ROP /ML /N,,;

MASM AA831RAC /ML /N,,;
```

When you link your AMX application, be sure to include your customized AMX ROM Access Module *AA831RAC.OBJ* (created above) in your system link specification file.

The AMX ROM is then linked with the Microsoft linker as follows.

```
LINK AA831ROP+AA831ROS,AMXROM /MAP /NOE,AMXROM,AMX831.LIB;
```

This example generates file *AMXROM.EXE* in DOS executable file format. This execute file must be converted to a ROM image. The manner in which this is accomplished will depend upon the development tools which you are using for that purpose.

Microsoft CodeView

The Microsoft DOS and Windows® based CodeView Debugger supports remote, source level debugging of your AMX 86 system. Note that CodeView v4.10 provided with Visual C/C++ v1.5 no longer supports remote debugging and hence, is not suitable for testing AMX 86 applications.

To use releases of the Microsoft CodeView Debugger which support remote debugging, you must install the Microsoft remote monitor in your target hardware. Instructions for doing so are provided in the CodeView Reference Manual.

Use the AMX Breakpoint Manager to ensure that all task activity halts when you encounter a breakpoint. To include the Breakpoint Manager in your configuration using the AMX Configuration Manager, go to the Breakpoint Parameter window and check the box to enable the Breakpoint Manager. Set the breakpoint entry delay to 2 and the exit delay to 20.

If you use the AMX 86 PC Supervisor, be sure to include the PC Supervisor Debug Module so that both you and the debugger can share DOS and the PC devices. To include the PCS Debug Module in your configuration using the AMX Configuration Manager, go to the PC Supervisor Parameter window and check the box labeled "Debug Module required".

You should be aware that the debugger may make DOS calls while it traces your code. Because of this, you may encounter interference with your application's use of DOS and the PC devices.

> Note
>
> The Qualitas *386MAX* Memory Manager must not be resident on the host PC when you use CodeView v4.xx in the remote mode.

InSight Discontinued

The DOS-based InSight™ Debug Tool is no longer supported. It has been superceded by the AMX 86 *KwikLook* Fault Finder.

Existing InSight users can continue to use InSight with AMX 86. You must follow the directions for linking and debugging provided in earlier releases of the AMX 86 Tool Guide and in the InSight Reference Manual.

To use the updated AMX Configuration Builder for Windows with InSight, go to the Configuration Manager's Edit menu and select Preferences... from the menu. Check the box labeled "Show InSight configuration page". Then exit from the Configuration Manager and restart the manager. The InSight Parameter window will then be accessible via the InSight tab. There has been no change to the use and meaning of the InSight parameters presented in that window.

AMX 86 Configuration Extensions

If you use the AMX Breakpoint Manager, your AMX System Configuration Module will include the necessary breakpoint delay parameters. However, you may wish to revise the *...EXT* extension directive in your AMX User Parameter File to add the features described below. The extension is also used to provide additional PC Supervisor options.

The breakpoint delays and PC Supervisor options can be edited using the AMX Configuration Builder. In rare cases, you may have to alter extension parameters that are not handled by the Configuration Builder. To do so, find and edit the following statement in your User Parameter File *SYSCFG.UP*.

```
...EXT 0,0,DBGA,BPED,BPXD,PCSOPT
```

where:

0,0 = Reserved

DBGA = 0 = Debugger attributes
Bit 0 = 0 = reserved
Bit 1 = 1 if using a debugger which, at breakpoints, takes over the DOS INT 21H software interrupt and does not restore it upon continuation.
Bits 2 to 31 = 0 = reserved

BPED = n = Breakpoint entry delay (0 < *n* < 32768 AMX ticks)
Use *n = 2* for Microsoft.

BPXD = n = Breakpoint exit delay (0 < *n* < 32768 AMX ticks)
Use *n = 20* for Microsoft.

PCSOPT = PC Supervisor extension options
Bit 0 = 1 if the PC Supervisor driver for the Parallel Port Adapters should pass on all *INT 17H* requests for I/O on undefined or unused *LPTn* devices to the original BIOS handler.

You can use this option to force the PPA driver v3.02 or later to behave in the same manner as earlier versions.
Bits 1 to 31 = 0 = reserved

For example, when using the Microsoft tools, your User Parameter File *SYSCFG.UP* to should contain the following statement.

```
...EXT 0,0,0,2,20,0
```

If you edit the *...EXT* directive, use the Configuration Builder to generate a new copy of your System Configuration Module *SYSCFG.ASM*. Be sure to use the most recently issued version of the AMX Configuration Template File *AM831CG.CT*.

5. WATCOM (WC) Tool Guide

AMX™ 86 has been developed on a PC with Windows® NT v4.0 using the Microsoft and WATCOM software development tools listed below. The AMX libraries and object modules on the product disks have been generated using the most recent tools listed. If you are not using this toolset, you may have to rebuild the AMX libraries in order to use your out-of-date tools.

		v8.5	v10.5	v10.6	v11.0
MASM	Microsoft 80x86 Assembler	v5.1	v6.11d	v6.11d	v6.11d
WCC	C/C++ compiler	v8.5c	v10.5	v10.6	v11.0
WLIB	Librarian	v2.4	v10.5	v10.6	v11.0
WLINK	Linker	v7.0	v10.5	v10.6	v11.0
VIDEO	C Source Debugger	v3.1			
WD	C Source Debugger		v10.5	v10.6	v11.0

AMX 86 has been tested on the following platforms.

80x86 PC with MS-DOS v3 to MS-DOS v6.22
AMD Net186™ Demonstration Board
VAutomation *i*CON186™ TIPS3 Evaluation Board

Environment Variables

Set the following DOS environment variables to provide access to all AMX, Microsoft and WATCOM tools, header files, object files and libraries.

AMXPATH	Path to AMX installation directory (*...\AMX831*)
PATH	Path to AMX, Microsoft and WATCOM executable programs
WATCOM	Path to WATCOM installation directory (v10.0 and later)
INCLUDE	Path to all WATCOM include header files
WCC	WATCOM C command line switches
LIB	Path to all WATCOM object files and libraries
TMP	Path to a temporary directory for use by tools

Register Usage

The WATCOM version of AMX makes the following C interface register assumptions. Registers *AX*, *CX* and *DX* can always be altered by C procedures. Registers *BX*, *SI*, *DI*, *BP*, *SP* and all segment registers are preserved by AMX according to the WATCOM rules for C procedures. Integers are returned from C procedures in register *AX*. Longs and pointers are returned from C procedures in register pair *DX:AX*. The *DS* register is dedicated for access to global data in segment *DGROUP*. You must NOT use any C compilation switch which changes these register assumptions.

Stack Checking

The WATCOM C Compiler generates a runtime stack check at the entry point to every C function. The check verifies that, after local variable storage has been allocated on the stack, stack still remains available for use.

This stack checking is a compilation option which can be inhibited with the */s* switch during compilation (see WATCOM *WCC* switch options). 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 functions, you will have stack checking in effect.

In order to defeat WATCOM's runtime stack checking, KADAK provides an alternate stack check procedure in object module *AJ831WC.OBJ*. This module must be linked into your system prior to linking the WATCOM C Runtime Library. Stack checking is completely inhibited by this new module.

> Warning!
>
> Do not forget to link object module *AJ831WC.OBJ* with your application to inhibit run time stack checking by functions in WATCOM's library.

WATCOM Optimization

Note that the degree of optimization performed by the WATCOM compiler is very dependent upon the amount of memory available for use by the compiler. For this reason, you may find that the code generated for a particular module varies if the module is compiled at different times or on different machines depending on the memory configuration at the time of the compilation. Therefore, when you are hunting for program bugs (especially timing problems), be sure that the code under test is not inadvertently being modified by this compiler feature.

Making Libraries

To make a library from a collection of object modules, create a library specification file *YOURLIB.LBM*. Use the WATCOM version of the AMX library specification file *AMX831.LBM* as a guide. Make your library as follows.

```
WLIB @YOURLIB.LBM
```

Assembling the AMX System Configuration Module

Your AMX System Configuration Module *SYSCFG.ASM* is assembled using Microsoft's *MASM* assembler as follows. All AMX header files *AMX831xx.DEF* must be present in the current directory together with file *SYSCFG.ASM*.

```
MASM SYSCFG /ML /N,,;
```

Using the WATCOM C/C++ Compiler

All AMX header files *AMX831xx.H* must be present in the current directory together with your source file being compiled.

By default, the WATCOM compiler passes function parameters in registers wherever possible. It adds trailing underscores to function names and leading underscores to variable names. These conventions are incompatible with the AMX C interface. Fortunately, WATCOM supports the *cdecl* keyword which defines the C parameter passing convention with which AMX is compatible. The AMX keyword *AMCCPP* is defined to be *cdecl* forcing all AMX procedures and AMX-callable application procedures to pass parameters on the stack.

It is important to note that all application functions which are called by AMX must also be declared with the *cdecl* keyword. All tasks, Restart and Exit Procedures, Timer Procedures and Task Termination Procedures must be declared with the *cdecl* keyword. So must your User Error Procedure, Fatal Exit Procedure, Time/Date Scheduling Procedure and Memory Assignment Procedure.

By default, the AMX C header file *AMX831CF.H* assumes that the Microsoft compiler is being used. Therefore, you must define symbol *AMCCIF=11* on the compiler command line to force the AMX header files to select the WATCOM compiler characteristics.

Use the following compilation switches when you are compiling modules for use in the AMX environment.

```
by default          ; pass all parameters on stack and
                    ; add leading underscores to names
                    ; of procedures declared cdecl
by default          ; add leading underscores to variable names
by default          ; output object module FILENAME.OBJ
/DAMCCIF=11         ; AMX headers select WATCOM
/ml                 ; use Intel Large model but peg DS to DGROUP
/0                  ; use 8086/8088 instructions
/s                  ; no stack checking
/zu                 ; SS != DGROUP
/zdp                ; peg DS to DGROUP
/Zp1                ; pack structures
/d2                 ; (optional) generate debug information for
                    ; debuggers WD (or VIDEO)
```

The compilation command line is therefore of the form:

```
WCC FILENAME.C /ml /0 /s /zu /zdp /zpl /AMCCIF=11
```

You may find it convenient to establish these compilation options via the WATCOM environment variable *WCC* as follows. Note the use of # instead of = in the */d* switch.

```
C:>SET WCC=/ml /0 /s /zu /zdp /zp1 /dAMCCIF#11
```

Linking with the WATCOM Linker

The modules which form your AMX 86 system must be linked in the following order.

```
AJ831PRF.OBJ        ; AMX Segment Prefix Module
SYSCFG.OBJ          ; AMX System Configuration Module

Your MAIN module
Other application modules

AMX PC/AT or equivalent clock driver:
CH8253T.OBJ         ; AMX 8253 clock driver or your equivalent
or
CH_PC_T.OBJ         ; AMX PC BIOS clock driver


The following KwikLook object modules (only if KwikLook Manager used)
BJ840STB.OBJ        ; KwikLook Stub module
BJ840DRV.OBJ        ; KwikLook Device Driver
CHnnnnnS.OBJ        ; AMX chip-specific serial driver or your equivalent


AJ831WC.OBJ         ; AMX WATCOM Interface
AA832LWC.OBJ        ; (only if creating a DOS resident system)
                    ; (see Chapter 6.1 in PC Supervisor Reference Manual)

AA831BKE.OBJ        ; AMX Breakpoint exclusion module
                    ; (see Chapter 13.6 in AMX 86 User's Guide)

AJ831CV.OBJ         ; AMX 86 v2 to v3 conversion module
                    ; (only if converting an AMX 86 v2 application)

AA831RAC.OBJ        ; AMX ROM Access Module (customized)
                    ; (only if AMX placed in a separate ROM)
                    ; (see Appendix E of this manual)

AMX832.LIB          ; AMX 86 PC Supervisor Library
                    ; (only if PC Supervisor or its drivers are used)

AMX831.LIB          ; AMX 86 Library

WATCOM C Runtime Libraries for target hardware
```

Create a link specification file *YOURLINK.LKS*. Use the WATCOM version of the AMX 86 Sample Program link specification file *AM831SAM.LKS* or the AMX PC Supervisor Test Program link specification file *AMX832T.LKS* as a guide. If you are using the *KwikLook* Manager, use the WATCOM version of the *KwikLook* Sample Program link specification file *BJSAMPLE.LKS* as a guide.

> Note
>
> If you decide to omit any of the link commands which are in the sample link specifications, you may encounter link errors or run-time faults.

Link with the WATCOM linker using the following linker options on the command line or in the link specification file.

```
FORMAT DOS              ; generate DOS executable format
                        ; other formats can be generated
                        ; (see WATCOM manual)
NAME YOURLINK.EXE       ; generate executable file YOURLINK.EXE
DEBUG ALL               ; add debug information for WD (or VIDEO) use
```

The link command line is therefore of the form:

```
WLINK @YOURLINK.LKS
```

The resulting load module *YOURLINK.EXE* is suitable for use with the WATCOM WD (or VIDEO) debugger.

Linking a Separate AMX ROM

AMX 86 can be committed to a separate ROM as described in Appendix E of this manual. Copy the following files to the current directory.

AA831ROS.OBJ	AMX ROM Option Entry Module
AMX831 .LIB	AMX Library
AA831K .DEF	AMX Private Definitions
AMX831EC.DEF	AMX Error Code Definitions
AMX831RO.DEF	AMX ROM Option Definitions
AA831ROP.ASM	AMX ROM Option Module
AA831RAC.ASM	AMX ROM Access Module

Edit the copy of the AMX ROM Option Definitions file *AMX831RO.DEF* to define your ROM option specifications.

Assemble the ROM Option and ROM Access Modules using the Microsoft *MASM* assembler as follows.

```
MASM AA831ROP /ML /N,,;

MASM AA831RAC /ML /N,,;
```

When you link your AMX application, be sure to include your customized AMX ROM Access Module *AA831RAC.OBJ* (created above) in your system link specification file.

The AMX ROM is then linked with the WATCOM linker as follows.

```
WLINK FILE AA831ROP,AA831ROS LIBRARY AMX831.LIB NAME AMXROM.EXE
```

This example generates file *AMXROM.EXE* in DOS executable file format. This execute file must be converted to a ROM image. The manner in which this is accomplished will depend upon the development tools which you are using for that purpose.

WATCOM WD (or VIDEO) Debugger

The WATCOM DOS based WD (or VIDEO) Debugger supports source level debugging of your AMX 86 system.

To use the WATCOM WD Debugger in its remote mode, you must install the WATCOM server in your target hardware. Instructions for doing so are provided in the WATCOM Debugger User's Guide.

Use the AMX Breakpoint Manager to ensure that all task activity halts when you encounter a breakpoint. To include the Breakpoint Manager in your configuration using the AMX Configuration Manager, go to the Breakpoint Parameter window and check the box to enable the Breakpoint Manager. Set the breakpoint entry delay to 2 and the exit delay to 20.

If you use the AMX 86 PC Supervisor, be sure to include the PC Supervisor Debug Module so that both you and the debugger can share DOS and the PC devices. To include the PCS Debug Module in your configuration using the AMX Configuration Manager, go to the PC Supervisor Parameter window and check the box labeled "Debug Module required".

WD Debugger Instruction Tracing

The WATCOM Debugger provides a trace capability. Unfortunately the debugger may enable interrupts while tracing thereby making it impossible to trace through ANY critical region of code during which interrupts must be inhibited. This applies to your code as well as AMX code.

> Note
>
> You MUST NOT try to trace through ANY AMX code or you will experience bizarre effects (if you survive at all)

You should be aware that the debugger may make DOS calls while it traces your code. Because of this, you may encounter interference with your application's use of DOS and the PC devices.

WD Debugger and AMX Hardware Conflicts

The AMX Breakpoint Manager and PC Supervisor interfere with the debugger's use of the clock and serial ports. To overcome these incompatibilities, KADAK has provided a Restart Procedure *AABKRDB*. This procedure forces software clock interrupts to be serviced at ISP level and allows the serial interrupt (*IRQ3* or *IRQ4*), if enabled by the debugger, to remain enabled at breakpoints.

The WATCOM Debugger needs the clock software interrupt (*INT 1CH*) to occur at ISP level, not task level as normally provided by the PC Supervisor. Restart Procedure *AABKRDB* resolves this problem.

When used in its remote mode, the debugger expects to have complete use of one serial port. The debugger may enable serial interrupts in order to meet its communication timing requirements. However, by default, the AMX Breakpoint Manager inhibits all interrupts except the clock and keyboard at breakpoints. Restart Procedure *AABKRDB* resolves this problem.

> Note
>
> To use the WATCOM Debugger with AMX, you must include *AABKRDB* in your list of Restart Procedures.
>
> Using the AMX Configuration Manager, go to the Launch/Shutdown window and add *AABKRDB* to your list of Restart Procedures.

Note that during testing of AMX 86 with WATCOM's WD Debugger v4.0 in a remote serial configuration, the debugger was found to hang the host PC with a blank screen unless all breakpoints were removed from the target before quitting from the debugger.

WATCOM Graphics

Some functions, such as *_floodfill()*, in the WATCOM graphics library are not compatible with AMX since they expect the current stack to be the main program stack.

To use the WATCOM graphics library, an AMX task must be a Medium model task (see Chapter 14.7 of the AMX Reference Manual) with a fairly large stack (1Kb or more). Only one task can use the graphics library. You must include the AMX module *AJ831WC.OBJ* (v3.03 or later) in your link specification.

If you are using a version of WATCOM C/C++ prior to v9.01c, you must also include the following code fragment at least once in the task prior to calls by the task to the graphics library:

```
extern int _STACKLOW;
int _GetStackLow(void);

_STACKLOW = _GetStackLow();
```

InSight Discontinued

The DOS-based InSight™ Debug Tool is no longer supported. It has been superceded by the AMX 86 *KwikLook* Fault Finder.

Existing InSight users can continue to use InSight with AMX 86. You must follow the directions for linking and debugging provided in earlier releases of the AMX 86 Tool Guide and in the InSight Reference Manual.

To use the updated AMX Configuration Builder for Windows with InSight, go to the Configuration Manager's Edit menu and select Preferences... from the menu. Check the box labeled "Show InSight configuration page". Then exit from the Configuration Manager and restart the manager. The InSight Parameter window will then be accessible via the InSight tab. There has been no change to the use and meaning of the InSight parameters presented in that window.

AMX 86 Configuration Extensions

You must use the AMX Breakpoint Manager when using either WATCOM debugger. Your AMX System Configuration Module must include the necessary breakpoint delay parameters. You must then revise the *...EXT* extension directive in your AMX User Parameter File to add the debugger attribute described below. Note that the extension is also used to provide additional PC Supervisor options.

The breakpoint delays and PC Supervisor options can be edited using the AMX Configuration Builder. When using the WATCOM debuggers, you must alter the extension parameters that are not handled by the Configuration Builder. To do so, find and edit the following statement in your User Parameter File *SYSCFG.UP*.

```
...EXT 0,0,DBGA,BPED,BPXD,PCSOPT
```

where:

0,0 = Reserved

DBGA = 0 = Debugger attributes
Bit 0 = 0 = reserved
Bit 1 = 1 if using a debugger like WATCOM's WD (or VIDEO) which, at breakpoints, takes over the DOS INT 21H software interrupt and does not restore it upon continuation.
Bits 2 to 31 = 0 = reserved

BPED = n = Breakpoint entry delay (0 < *n* < 32768 AMX ticks)
Use *n = 2* for WATCOM.

BPXD = n = Breakpoint exit delay (0 < *n* < 32768 AMX ticks)
Use *n = 20* for WATCOM.

PCSOPT = PC Supervisor extension options
Bit 0 = 1 if the PC Supervisor driver for the Parallel Port Adapters should pass on all *INT 17H* requests for I/O on undefined or unused *LPTn* devices to the original BIOS handler.

You can use this option to force the PPA driver v3.02 or later to behave in the same manner as earlier versions.
Bits 1 to 31 = 0 = reserved

When using the WATCOM tools, your User Parameter File *SYSCFG.UP* to must contain the following statement.

```
...EXT 0,0,2,2,20,0
```

After you edit the *...EXT* directive, use the Configuration Builder to generate a new copy of your System Configuration Module *SYSCFG.ASM*. Be sure to use the most recently issued version of the AMX Configuration Template File *AM831CG.CT*.

This page left blank intentionally.

Appendix A. Building AMX 86

The AMX 86 Libraries are provided by KADAK ready for use with the toolsets supported by KADAK. There should be no need to remake the AMX 86 Libraries unless you have altered the AMX 86 source code or are using an out-of-date toolset which requires a rebuild of AMX for backwards compatibility.

Let *XX* be the toolset id for the toolset combination for which you wish to remake AMX 86. See the list of supported toolset ids in Chapter 1.

It is assumed that AMX 86 has been installed into directory *...\AMX831* ready for use with toolset *XX*.

To construct the AMX Library, you must first open a Windows Command Prompt window. Go to AMX directory *...\AMX831\MAKE* and examine the header in batch file *EN831XX.BAT*. You must set the environment variables specified in that file so that all of the required toolset files and AMX 86 files are accessible.

To set the environment ready to construct AMX 86 using toolset *XX*, you must specify the version number for each tool which forms part of the toolset. In the following example, the version numbers are given the names *asmver*, *cver* and *dbgver* for illustration purposes. Use the values which correspond to the versions of the tools which you are using. Allowable values are shown on the next page. For some toolsets, one or more of the parameters may not be required and must therefore be omitted.

Use batch file *EN831XX.BAT* to establish the environment as follows:

```
...>EN831XX asmver cver dbgver
```

Finally, you will need a *MAKE* utility such as Borland *MAKE.EXE* or Microsoft *NMAKE.EXE*. Your *PATH* environment variable must provide access to this utility. AMX file *AM831XX.MAK* is a make specification file that purposely avoids constructs and directives that tend to vary among make utilities. You may have to edit this file to meet the requirements of your particular make utility.

Go to AMX directory *...\AMX831\MAKE* and delete all header files (**.DEF* and **.H*), if any, left in the directory from previous sessions. Then issue the following command. Be sure to replace the "*C:\KADAK*" in the command with your AMX 86 *drive:\path* specification.

```
! Make AMX 86 using Borland MAKE
...>MAKE -DTOOLSET=XX -DAMXPATH=C:\KADAK\AMX831 -fAM831XX.MAK

! Make AMX 86 using Microsoft NMAKE
...>NMAKE TOOLSET=XX AMXPATH=C:\KADAK\AMX831 /F fAM831XX.MAK
```

When the make is complete, directory *..\TOOLXX\LIB* will contain your updated AMX 86 Libraries and object modules.

Directory *..\ERR* will contain a summary of error messages, if any, produced during the make process.

Note that if you add the Borland *-n* switch (Microsoft */n* switch) immediately following the *MAKE* (*NMAKE*) directive on the command line, the make utility will list the make operations on the screen but will not actually do the make. This can be helpful in locating path problems if you have not properly installed AMX 86 or have not provided correct environment variables.

The following tool versions are currently supported for each of the supported toolsets.

Toolset id	**Assembler asmver**	**C/C++ cver**	**Debugger dbgver**
PD	5.0	5.0	5.0
	6.0	6.0	6.0
PX	6.0	6.0	6.0
TC	1.0	2.0	1.0
	3.1	4.02	4.02
	4.0 (Note 1)	4.52	4.6
	5.0 (Note 1)	5.0	5.0
MC	5.1	5.1	
	6.0	6.0	
	6.0	7.0	
	6.11	++1.5 (Note 2)	
	6.11d	++1.52 (Note 2)	
WC	5.1 (Note 3)	8.5	
	6.11d	10.5	
	6.11d	10.6	
	6.11d	11.0	

Note 1: The assembler provided with Borland Turbo Assembler v4.0 or v5.0 identifies itself as version 4.1.

Note 2: The compiler provided with Microsoft 16-Bit Visual C++ v1.5 and v1.52 identifies itself as version 8.00c.

Note 3: The Microsoft *MASM* assembler is used with WATCOM toolset *WC*.

Example: Make AMX 86 for toolset *PD*.

```
...>EN831PD 6.0 6.0 6.0
...>MAKE -DTOOLSET=PD -DAMXPATH=C:\KADAK\AMX831 -fAM831PD.MAK
```

Common Problems

When rebuilding the AMX libraries, a number of Windows related or *MAKE* dependent problems may be encountered.

Your *MAKE* utility must be able to issue simple commands such as *COPY* and *ERASE*. It must also be able to invoke the batch file *AM831XX.BAT* which is used to run the assembler, C compiler, linker/locator and librarian for toolset *XX*.

Some *MAKE* utilities are provided in different forms for use in different environments. Choose the simplest version which can be executed within the Windows Command Prompt window. For example, Borland offers a real mode *MAKE* and a protected mode *MAKE* with some form of built-in DOS extender. The protected mode version has been observed to fail, in some DOS environments, when it tries to allocate memory in order to execute a batch file such as *AM831XX.BAT*.

The make process has only been tested in a standard Windows Command Prompt environment. Although the make process is designed to operate correctly under stand-alone DOS, this mode of operation has not been tested by KADAK and is not supported by KADAK.

So what is a standard Windows Command Prompt environment? It is the configuration of Windows on your PC which allows your *MAKE* utility to run batch file *AM831XX.BAT* and still have enough memory free to use the assembler, C compiler, linker/locator and librarian for toolset *XX*. Try to start with at least 500K of memory available.

RAM drives and temporary disk storage can also be problems. If all of your extended memory is used for a RAM drive, there may not be enough memory free for use by the *MAKE* utility and the software tools. If the drive specified by your *TMP* or *TEMP* environment variable for use for temporary files is almost full, compilations or links may fail. In the past, some tools have been observed to crash or hang if they run out of memory or disk space. Sad but true!

The Windows command line length imposes a restriction which may affect the construction process if your *AMXPATH* environment variable specifies a long path string. For example, if you install AMX in directory *D:\PROJECT\YOUR86*, then the build process will use *AMXPATH* to reference source and object files like:

```
D:\PROJECT\YOUR86\AMX831\AMX\AA831KA.ASM
D:\PROJECT\YOUR86\AMX831\TOOLXX\LIB\AA831KA.OBJ
```

Obviously, if both of these strings appear in a single command within the batch file *AM831XX.BAT*, the command will probably fail.

If you suspect that this problem is occurring, use the *SUBST* command to substitute a single drive letter, say *Z:*, for the path string *D:\PROJECT\YOUR86* as in the following example.

```
SUBST Z: D:\PROJECT\YOUR86
SET AMXPATH=Z:
```

This page left blank intentionally.

Appendix B. Trouble Shooting (Application Notes)

Most of the commonly asked questions about AMX are answered in the AMX User's Guide. However, finding the answer may sometimes be difficult. The following suggestions will hopefully guide you in the right direction.

Following these suggestions are a set of AMX Application Notes. These notes address subjects or problems that are not directly covered elsewhere in the AMX manual.

Sample Program

The AMX Sample Program, a simple AMX application, is provided ready for use on each of the target platforms on which AMX has been tested. The Sample Program is described in the AMX C Programming Guide.

The Sample Program source code for toolset *XX* is in AMX installation subdirectory *SAMPLE*. Other required modules (serial drivers and clock drivers) are also installed in subdirectory *SAMPLE*.

Be sure to read batch file *AM831SAM.BAT* in directory *TOOLXX*. It describes the pieces needed to build the Sample Program. You must also be familiar with the commands needed to compile, assemble and link/locate the application (see the next topic).

Using your Tools

If you are not sure which software development tools you should use, read Chapter 1 of this Tool Guide. It describes each toolset combination with which AMX has been tested and assigns a mnemonic, *XX*, to each.

If you are having difficulty with software development toolset *XX*, read the specific Tool Guide for toolset *XX*. The guides are provided as separate chapters in this overall Tool Guide.

In particular, you need to be able to do the following:

Assemble the AMX System Configuration Module
Compile and/or assemble your application modules
Link (and locate) your application load module

Building the AMX Library

There should be no need to rebuild the AMX Library. The library and related object modules for use with toolset *XX* are provided in the AMX installation subdirectory *TOOLXX\LIB*.

If you must rebuild the AMX Library (because you are using an out-of-date toolset or because you just need to confirm that you can do it), follow the instructions in Appendix A of this Tool Guide.

Linking Problems

Most problems arising in the actual construction of an embedded AMX application are linker/locator related.

To start with, be sure that you understand the memory layout and addressing constraints of the target hardware that you are using. Be sure that your link commands meet the requirements. For examples, see the sample toolset *XX* Link (and/or Locate) Specification File in AMX installation subdirectory *TOOLXX*. Note that these files match the hardware configuration of the boards used by KADAK. Your board may differ!

Check the order in which your object modules and libraries are linked. The order must be as defined in the toolset *XX* Tool Guide. Watch this one! There may be subtle differences from one toolset to the next.

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 sample Link Specification File for toolset *XX* in the board specific subdirectory of AMX installation subdirectory *TOOLXX\SAMPLE*.

AMX™ 86 Tool Guide (Appendix B)

Application Note 1

This technical note is provided to summarize answers to the most frequently asked questions concerning the use of AMX 86. The topics covered are:

- AMX and PC Supervisor Caveats
- Debugging AMX Systems
- ROMing C Code
- Large vs Medium Memory Models
- DOS Memory Management and *malloc*
- Using Microsoft Tools
- Local Networking Issues

B1.1 AMX and PC Supervisor Caveats

Task Trigger vs Message Queuing

Do not mix *ajtrig* and *ajsend* (or *ajsenw*) calls to the same task. Tasks with no mailboxes are triggered using *ajtrig*. Tasks with mailboxes are expected to receive messages sent using *ajsend* or *ajsenw*.

If you mix *ajtrig* and *ajsend* calls to a task, the task will execute once in response to each *ajtrig* call but will not receive a message on its stack. Furthermore, unless the task decodes the AMX message extension, it has no way of determining that this is the case and can therefore only assume that the garbage on its stack is a message which it will process with unpredictable results.

Message Envelopes

If you use the Event Manager or the PC Supervisor you must provide some message envelopes for their use. A minimum of ten (10) envelopes is recommended.

If you have many events being signalled concurrently from ISPs in your application, you may have to increase the number of available envelopes. The Event Manager uses one envelope each time an event is signalled by an ISP.

Using Third-Party Subroutine Libraries

Many AMX applications require your use of specialized third-party subroutine libraries for database access, screen access, graphics, etc. These subroutine 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.

Database packages in particular make DOS calls for file I/O access. It is best to use the PC Supervisor to reserve and release DOS around your calls to the database package.

DOS Calls

When a task makes a DOS INT 21H function call, the task is suspended. The actual DOS call is made by the PC Supervisor Task which is usually of lower priority than all application tasks.

Consequently, all active tasks which are of lower priority than the task which initiated the DOS call but of higher priority than the PC Supervisor Task will execute before the actual DOS request can be handled by DOS. This fact may lead you to conclude that your DOS calls are taking a long time to be honored. That is true but only because you have other higher priority active tasks present in your system.

DOS Command Task

When using the DOS Command Task in your resident AMX application, remember that resident tasks must NEVER change the current directory. Your resident tasks must ALWAYS use full path names for file access.

DOS Command Task Loader *AA832LD.EXE*

The PC Supervisor DOS Command Task program loader requires a valid copy of file *AA832LD.EXE*. This file is actually modified by the loader while loading a DOS program. If the copy of file *AA832LD.EXE* is ever corrupted, then the DOS Command Task may not be able to load a DOS program or may cause it to be loaded incorrectly.

Under normal operation no corruption will occur. However, while you are creating and testing your resident AMX system, you may encounter such corruption.

For example, suppose that a DOS program is being loaded and the file *AA832LD.EXE* has been partially altered just as an application device interrupts. A previously undetected bug in the application causes a fatal error and the processor halts, hangs or otherwise crashes. You reboot DOS and now when you start your AMX system, you cannot even load a program because file *AA832LD.EXE* is corrupt.

The solution is to keep a copy of the original file *AA832LD.EXE* provided with AMX with name *AA832LD.SAV*. Edit your *AUTOEXEC.BAT* file to copy this file to new file *AA832LD.EXE* in the directory which you specified in your installation of the PC Supervisor DOS device driver in your *CONFIG.SYS* file.

PC Supervisor Interrupt Nesting

The PC Supervisor saves and restores all of the 8 (or 16) hardware interrupt vectors in order to make all of the PC devices adhere to the AMX nesting rules. This feature has an effect upon the order in which the hardware interrupt vectors are saved and restored.

If you are going to take over a hardware interrupt vector for your private use, you may do so in a task or in a Restart Procedure.

When you shut down your AMX system, you can restore the hardware interrupt vector in a task before calling *ajexit* or in an Exit Procedure which will be executed by AMX after the *ajexit* call. The PC Supervisor will not be shut down until all of your Exit Procedures have been executed.

When you take over a hardware interrupt, be aware that the previous vector pointed to the conforming AMX Interrupt Service Procedure installed by the PC Supervisor, not to the original PC handler.

If you intend to chain into an original PC hardware interrupt chain, you must read the interrupt vector before you launch AMX. Do not use *ajivtr* to read the vector. Then your Restart Procedure or task can install its own hardware ISP and chain to the saved copy of the original PC handler.

C Code Portability

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

Make all AMX task and object identifiers be of type *AMXID*.

Use a *typedef* to define *AMXTVAL* to be an AMX timer value. Then cast constants to be of type *AMXTVAL* when passing timer values to AMX.

```
typedef long AMXTVAL;
ajwatm((AMXTVAL)2000);                    /* Wait 2 seconds          */
```

Use only the least significant 16 event flags of each event group. This will permit designs for 32-bit processors (80386, 68000 etc.) to be readily ported to 16-bit processors (8086 etc.).

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

If you use the keyword *cdecl* when declaring public functions, it will ease porting C code from one C compiler to another. See the AMX Sample Program listing in the AMX C Programming Guide or examine source module *AM831SAM.C* for an example. Note that the manner in which functions such as *main()* are declared will depend upon your version of C. For simplicity, the use of *cdecl* is omitted throughout the AMX User's Guide.

B1.2 Debugging AMX Systems

Breakpoints and Tracing

When using a debugger on your AMX system, it is important to be aware of subtle timing effects which may occur. This is especially true of a source level debugger such as Microsoft's CodeView where it is easy to trace across function calls.

When the AMX Breakpoint Manager and PC Supervisor Debug Module are linked into your system, AMX always gains control when a breakpoint is taken. AMX stops all timing activity by temporarily ignoring the clock. All tasks except the PC Supervisor tasks and the task in which the breakpoint occurred are temporarily suspended.

When AMX detects that your breakpointed task has resumed execution, it resumes timing operations and restores the state of all tasks which were suspended at the breakpoint. Unfortunately, there is no easy way for AMX to determine that your debugger has relinquished control and allowed your code to resume execution. The AMX Breakpoint Manager monitors task switches and clock interrupts to detect your task's resumption.

If you trace single instructions (not whole C statements), the debugger never actually gives up control. Therefore, while you are single stepping past a breakpoint, your AMX system remains temporarily shut off.

If you allow your system to run and hit another breakpoint several instructions or statements beyond the first breakpoint, it is probable that AMX will not have even detected the program resumption before the second breakpoint is encountered. Therefore you will hit your second breakpoint with no intervening task activity other than that of the task you are debugging.

For example, if you use CodeView to trace over a call to send a message to another task, the other task cannot run unless it is of higher priority. It will run, of course, as soon as AMX detects the end of the breakpoint and resumes normal activity. As long as you are aware of this property, your debugging should proceed smoothly.

Thus, once you hit a breakpoint, real-time stops. It only resumes when you allow your system to free-run again.

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. The common complaint is: "I can only put breakpoints in tasks but I never get to any of my task code!"

Startup is no-man's land. It is a grey area where DOS (or your loader) has given your program (i.e. AMX) control but AMX has not yet created a solid AMX environment.

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. For instance, only tasks can make DOS calls. Therefore, Restart Procedures cannot make DOS calls. It is for this reason that Restart Procedures cannot include `printf` statements to assist in debugging. (We know it works sometimes but not always.)

You can usually use your debugger to step through your own Restart Procedures although we can make no such guarantee. Do not try to step through the AMX procedures which your Restart Procedures call. Once your last Restart Procedure (yours, not the AMX or PC Supervisor Restart Procedures) has been called, you must let AMX free run. Any attempt to breakpoint your way through the remaining 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 doesn't, 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.
- You repeated some of the private AMX Restart Procedures in your list of Restart Procedures. The AMX Configuration Builder automatically includes all private AMX Restart Procedures which AMX needs. You only have to define your own or those which the Tool Guide instructs you to define.
- 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.
- You are using DEBUG, SYMDEB or CodeView to debug a system which includes the PC Supervisor Task but you have NOT included the PC Supervisor Debug Module in your system.

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 Sample Program code "borrowed" and adapted for a new application.

Using a Fatal Exit Procedure

Do not ignore the creation of a 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 provided a Fatal Exit Procedure and are not using the PC Supervisor, AMX will halt with interrupts disabled forcing you to initiate a power reset to recover. However, you can intercept this fatal shutdown by providing a 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 13.1 of the AMX User's Guide for the rules.

Suggestion: On a PC, use the default PC Supervisor Fatal Exit Procedure.

Using Borland's Turbo Debugger

The PC Supervisor Debug Module works with debuggers such as Borland's Turbo Debugger.

The Turbo Debugger is a particularly nasty debugger from the AMX perspective. It unhooks the PC Supervisor from the DOS INT 21H path thereby precluding our detection of the debugger's DOS calls. It also unhooks the PC Supervisor hardware clock ISP, further frustrating the Breakpoint Manager's operation.

All of these characteristics make it impossible to detect the exact instant at which the debugger allowed your AMX system to resume execution from a breakpoint. Consequently, all tasks which were halted when a breakpoint occurred will remain halted until the breakpointed task makes a DOS call, asks for keyboard input, ends or is otherwise suspended.

> Warning
>
> When the Turbo Debugger proceeds from a breakpoint, the 8259 interrupt mask has the clock interrupt enabled, even if your system had it disabled when your breakpoint was encountered.

80386 Debuggers

The AMX Breakpoint Manager and PCS Debug Module will operate with some debuggers which take advantage of the 80386 debug registers to provide real time watchpoint detection.

The PCS Debug Module is not compatible with 80386 PC debuggers which arbitrarily remap interrupt vectors 8 to 15 inclusive to avoid 80386 processor conflicts. AMX and PCS are unaware of this remapping and are therefore in direct conflict with the debugger.

Some 80386 debuggers, such as Borland's *TD386*, may use the 80386 virtual 8086 mode of operation. When operating in this mode, the AMX Breakpoint Manager is unable to detect breakpoints. Furthermore, the PCS device Interrupt Service Procedures do not support virtual mode. Consequently, the 80386 debugger may or may not be compatible with AMX and the PC Supervisor.

For example, Borland's *TD386* version 1.0 hangs while at a breakpoint as soon as it attempts to access disk. However, *TD386* version 2.0 has no such problem.

Debugging Caveats

Never use your debugger's *QUIT* command to leave your AMX system and return to DOS. Your AMX system must invoke *ajexit* to force an orderly AMX shutdown. When AMX attempts to return to DOS, the debugger will indicate that your program under test has terminated. Only then can you use your debugger's *QUIT* command to return to DOS.

Never use a debugger's *Ctrl-C* or *Ctrl-Break* command to try to stop a "run-away" AMX system. These mechanisms are not compatible with your multitasking environment and often lead to catastrophic failure. The debugger must only gain control via breakpoints, watchpoints or traces.

B1.3 ROMing C Code

Contrary to what some may say, most C compilers can be used to produce ROMable code. Here are a few guidelines to follow to successfully produce ROMable code. Note that these solutions may not be applicable to all AMX users. Much depends upon the constraints of your application.

Startup code

Most suppliers include the source for their C startup code so you can change it as you see fit. A more straight forward approach would be to avoid declaring a function called `main()`, so the startup code would not be automatically included. Usually a ROMed system requires specialized startup code anyway. When your startup code has finished processing, it can call a C function with a name other than `main()`.

Note that if you omit the startup code, some C library functions (especially file I/O) may not be usable since you have eliminated their internal initialization. Many ROM based systems are unaffected by this constraint since they require no DOS file support.

Static/external data access presents the biggest problem with ROMed code. The C language includes no built-in features to allow code to distinguish between ROM and RAM. When static data is defined in a C module it just becomes part of the data segment.

Constants are also placed in a data segment and hence are not automatically part of your ROM (code) image. You should also be aware that some C code generators occasionally optimize code by creating private constants (usually segment values) which are placed in the initialized data segment.

String constants are also placed in the initialized data segment instead of in the code or constant segment. All C compilers must do this to meet the C language specification which permits such strings to be altered.

If you simply locate your entire `.EXE` file at your ROM start address, all static data is placed in ROM and cannot be modified by your program. Your C startup code has to copy the initialized data from ROM to RAM.

Library Function Reentrancy

Microsoft and others have never claimed that all of their library functions are reentrant. When calling non-reentrant functions in a multitasking environment, you must protect the code to avoid problems. One approach is to be sure that only one task ever calls the function. If this is too restrictive, a resource semaphore may be used to control access to the function. Library functions which call DOS require the reservation of DOS using the PC Supervisor functions `ajpdrs` or `ajpdrp` to provide the necessary protection.

We have observed the following: Floating point and math routines are usually not reentrant. File handling, I/O and memory allocation functions (`fopen`, `fgets`, `printf`, `malloc`, etc.) are, in general, not reentrant and must be protected by calls to `ajpdrs` or `ajpdrl`. Most string manipulation and data conversion functions (`strcpy`, `strcmp`, `atoi`, `isalpha`, etc.) are reentrant and may be used safely anywhere. If in doubt, assume the worst.

Locating the EXE File

When you have linked your `.EXE` file, it must still be located properly and burned into ROM. Use one of the program locators available from vendors such as Paradigm.

Caveat

We cannot guarantee the suitability of your C code for real-time or ROMed applications. The hints given are observations, not documented features. There is nothing to prevent the supplier of your C compiler from changing the operation of their compiler at any time.

B1.4 Large vs Medium Memory Model

What is Large?

Intel's segmentation continues to plague users of the 80x86 family of processors. To compound matters, Microsoft and others have chosen to ignore Intel's model definitions.

Intel defines the Large model to provide separate code and data segments for each module and a separate stack segment for program execution. This is the Large model supported by AMX. Microsoft C does not support this model. That is why you must use the compilation switch `/Alfw` when compiling your C modules for use with AMX.

Intel defines the Medium model with `FAR` pointers to provide a separate code segment for each module and to merge data and stack into one segment. This is the Medium model supported by AMX. Microsoft C supports this model but calls it Large. Therefore, if you compile your C modules to be Large using the `/AL` compilation switch, you are really selecting the Intel and AMX Medium model.

> Note
>
> Large model tasks must not call Medium model procedures because the Large model task's stack is not part of group `DGROUP`.
>
> Medium model tasks can call Large model procedures as long as they ensure that pointers are passed as `FAR` pointers.

Microsoft C Runtime Library

The Microsoft Large model C runtime library is really a Medium model library. Some of the procedures in the library assume that registers *DS* and *SS* both point to the single data/stack segment. Unfortunately, when a Large model AMX task executes such a procedure, *DS* and *SS* do not match and the procedure fails, usually catastrophically.

Most simple C library functions, such as those used for string manipulation, do not depend on this *DS = SS* requirement. Unfortunately, there is no way to identify which functions do and which do not have the need.

Generally, it is the more complex, low level functions such as *_read* and *_nmalloc* which have been found to plague AMX Large model users.

The following is a list of Microsoft C 5.1 functions which may assume *DS = SS*. KADAK cannot guarantee the completeness or correctness of this list; it is provided for guidance only.

```
_exit                 scanf
abort                 spawn derivatives
chsize                sprintf
creat                 sscanf
cscanf                tmpfile
exec derivatives      vfprintf
exit                  vprintf
fprintf               vsprintf
fscanf
printf
```

Several solutions, none totally acceptable, are available.

Solution 1: All tasks making C library calls must be AMX Medium model tasks so that *DS = SS* at all times. The task stacks for all of these tasks reside in the single data/stack segment.

Solution 2: Any task which makes C library calls known to exhibit the problem must be an AMX Medium model task. Other tasks which call valid Large model library functions can be Large model AMX tasks.

Solution 3: Acquire the Microsoft C library source code and remove the Medium model dependency from its Large model library or the subset of the library which you need.

Mixed Model Programming

There is often confusion over the definition of the Medium model. AMX supports Intel's Medium model with extended (*FAR*) pointers. In this model, all *PUBLIC* (global) procedures are *FAR* procedures; that is, a 32-bit return address in segment:offset form is pushed onto the stack when the procedure is called.

All AMX procedures are *FAR* procedures. Any AMX procedure which expects a pointer as a parameter requires the caller to provide a *FAR* pointer. This requirement is consistent with the Medium model definition which supports *FAR* pointers.

Unfortunately, some high level language compilers have not adopted Intel's definition of the Medium model. They use a Medium model with separate code segments and one combined data, stack and memory segment as defined by Intel, but only offer *NEAR* pointers to data items. Their Medium model is therefore not compatible with AMX and should be avoided.

Consider the following example. Assume that a Medium model task calls the following procedure to send a simple integer message to another task at priority 3.

```
int sendmsg(unsigned int taskid, int message) {
        return(ajsend(taskid, 3, &message));
        }
```

If this procedure is compiled using the Medium model with Microsoft's C Compiler, the pointer *&message* is a *NEAR* pointer (16-bit offset only) to the variable *message* on the caller's stack. However, AMX procedure *ajsend* expects a 32-bit *FAR* pointer to the message. Consequently, AMX will receive an invalid message pointer and will fail to send the proper message to task *taskid*.

To properly code this procedure using the Medium model, you must use the keyword *FAR* to force *&message* to be a *FAR* pointer.

```
int sendmsg(unsigned int taskid, int message) {
        return(ajsend(taskid, 3, (char FAR *)&message));
        }
```

B1.5 DOS Memory Management and *malloc*

When a program is invoked by DOS, the program owns all available memory. The startup code of each of the supported C compilers will return the extra memory to DOS before calling your *main* program. The DOS memory allocation functions will operate properly on the remaining memory not required by your program. This means that C library memory routines, such as *malloc*, will work properly as will library functions, such as *fopen*, that call them.

Tasks cannot share *malloc* (or procedures which call *malloc*) unless you treat *malloc* as a serially reusable resource. The Semaphore Manager can be used for this purpose. Alternatively, you can use *ajpdrs* or *ajpdrl* to protect calls to these non-reentrant functions.

If you do not wish to use the C or DOS memory allocation functions or if your application is to run in a system without DOS, use the AMX Memory Manager if dynamic memory management is required.

Microsoft *malloc* Allocation Strategy

The process by which memory is allocated by Microsoft C 5.1 *malloc* is as follows. The C startup code expands its *DGROUP* data segment to 64K and returns all of the memory above that point to DOS. This free memory is referred to as the far heap. The memory above the stack segment to the end of the *DGROUP* segment is called the near heap.

When you call *malloc*, it tries to meet your request by allocating memory from the far heap. If it does not have enough memory in the far heap under its own control, it will request DOS for more memory until eventually your program once again owns all of the memory to the top of available memory. Then, and only then, *malloc* tries to meet its caller's request out of the near heap.

After several *malloc* calls, your memory image will appear as shown in Figure B1.5-1.

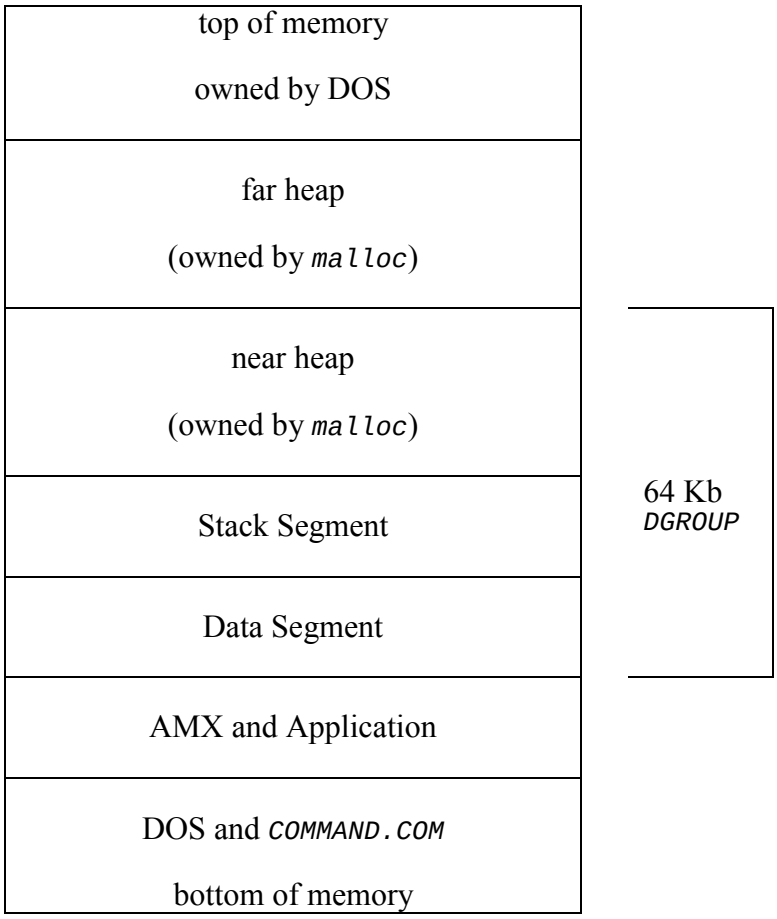

Figure B1.5-1 Microsoft C Memory Allocation

Near Heap Fault

Unfortunately, when *malloc* attempts to allocate memory from the near heap, it fails in an AMX system with catastrophic effects. It fails because the Microsoft Large model library near heap allocation functions assume that registers *DS* and *SS* will both reference the *DGROUP* segment, an Intel Medium model assumption.

This fault can be remedied by providing near heap allocation functions to eliminate those provided by Microsoft. KADAK has provided these replacement functions in module *AJ831MC.OBJ*. Include this module in your AMX system link. Don't forget to use the */NOE* link option if you are using a recent release of Microsoft's linker *LINK*.

Since the near heap is no longer used, you may wish to recover the memory allocated to it. To do this, use the Microsoft *EXE* File Header Utility *EXEMOD* to examine your AMX system execute file *AMXSYS.EXE*.

```
EXEMOD AMXSYS.EXE
```

One of the parameters listed in both hex and decimal is the minimum allocation in paragraphs (say *1CA* hex). Adjust the maximum allocation for your system to be the minimum as in the following example.

```
EXEMOD AMXSYS.EXE /MAX 1CA
```

Since the startup stack segment is only used to launch and shut down your AMX system, you should use the *LINK* or *EXEMOD* option */STACK* to also reduce the stack segment to a minimum.

Resident AMX System Memory Allocation

Resident AMX systems can, with care, also use their copy of *malloc*. The restrictions for shared access to *malloc* by multiple resident AMX tasks still apply.

When an AMX system goes resident, all of memory above some upper limit must be returned to DOS for use by the DOS Command Task to load and execute other DOS programs. The resident system load limit strategy used to determine this upper limit is as follows.

If any memory is already free above the resident AMX system, it is assumed that the C startup code has already properly sized the program and the upper limit of the resident system is set to match the current top of the far heap.

Alternatively, if all of memory is still owned by the resident system, then no C startup code was provided and the upper limit of the resident image is set to the bottom of the stack segment thereby freeing the startup stack segment and all memory above it.

With this strategy in place, a resident system can avoid memory fragmentation, yet arrange for its run-time memory allocation needs to be met as follows. Assume that your resident AMX system has a worst case memory allocation requirement of *MMAX* bytes. Include the following code sequence in your *main* C program prior to your AMX launch.

```
mp = malloc(MMAX);

if (mp != NULL) {
        if (malloc(1) == NULL)
              exit(errorcode);
        free(mp);
        }
```

B1.6 Using Microsoft Tools

AMX 86 is delivered ready for use with Microsoft's software development tools as described in the Microsoft (toolset *MC*) chapter of this tool guide.

AMX 86 remains code compatible with older versions of the Microsoft tools. However, you may have to assemble AMX and PCS source files and rebuild libraries *AMX831.LIB* and *AMX832.LIB* in order to use AMX with older tool sets.

You must be sure to use compatible tool sets. For instance, C code compiled with Microsoft C 6.0 may not link with the linker delivered to you with MASM v4.0.

> Warning
>
> Be sure to link AMX object module *AJ831MC.OBJ* with your application. Use the */NOE* linker switch in your link specification.
>
> Older versions of the C compiler may not support AMX function prototyping which, by default, is enabled.

B1.7 Local Networking Issues

If you require TCP/IP network support, you will find that KADAK's KwikNet™ 86 TCP/IP Stack is ready for use with AMX 86. No porting is required.

Although AMX does not provide local network support, you may still be able to use AMX with your local network software on PC hardware. You will need the PC Supervisor to arbitrate the use of DOS and to provide the file locking features required by the local network software.

Most local network software competes with the PC Supervisor for control of the clock. Successful use of AMX in your local network environment depends on the resolution of this conflict.

The Problems

Two sources of conflict exist: the hardware clock interrupt (IRQ0 on the master 8259 interrupt controller) and the software clock tick interrupt (INT 1CH).

There can only be one handler in charge of the PC hardware clock interrupt. That handler must remove the clock interrupt request, provide standard ROM BIOS timing services and issue the non-specific EOI to the 8259 interrupt controller to enable lower priority interrupts.

One of the services normally provided by the hardware interrupt handler is the generation of the PC's software clock tick interrupt (INT 1CH). It is this software interrupt which leads to serious problems in the AMX multitasking environment.

Resident DOS programs (TSRs) such as print spoolers may hook themselves into the INT 1CH chain and use the software tick as a periodic window of opportunity. For example, if the printer is ready and DOS is not in use, a print spooler may make a series of DOS calls to get more data to send to the printer. However, in your AMX system, this lengthy processing is occurring in an ISP to the detriment of your high priority tasks.

The PC Supervisor resolves this software clock tick problem by deferring generation of the software clock tick. The PCS clock driver services the hardware clock interrupt and leaves generation of the software clock tick interrupt to the PCS Clock Tick Task whose execution priority can be adjusted by you to meet the needs and constraints of your system.

Having solved one problem, the PC Supervisor may have created another one for time critical operations required by your network software. Since the PCS Clock Tick Task may be of lower priority than some of your high priority tasks, it may suffer variable length delays in its execution imposed by your tasks as they perform their higher priority services. This pseudo-random "jitter" of the software clock tick generated by the Clock Tick Task may prove unacceptable to your network software.

Service of the hardware clock interrupt is further complicated by the irrational 18.2 Hz clock frequency used on the PC. For real time control, the PC Supervisor allows the clock frequency to be set to 20 Hz (precise) or 100 Hz (not precise). The PCS clock driver then derives the 18.2 Hz software clock tick frequency for use by DOS, its device drivers and other resident software.

Software Clock Control

The PC Supervisor gives you direct control over the point of generation of the software clock tick interrupt (INT 1CH). Procedure *ajclop* (described later) can be used to force the software tick to be serviced in the hardware interrupt handler at ISP level instead of being deferred to the Clock Tick Task. Alternatively, it is possible to inhibit software clock tick service.

An example offers the easiest method to show the effects of these options in a real AMX system. The order in which handlers are hooked into the clock tick chain is crucial.

Assume that each of the following clock tick handlers are hooked into the INT 1CH software interrupt chain and that each passes the tick on.

PCTICK The head of the PC software clock tick chain when your AMX program is first started.

MAINTICK The handler, if any, which your main program installs at the head of the chain prior to launching AMX.

RRTICK The handler, if any, which one of your Restart Procedures installs at the head of the chain.

TASKTICK The handler, if any, which one of your tasks installs at the head of the chain.

For each of the three modes of operation, the order of execution of these handlers will be as illustrated in Figure B1.7-1. The handler at the top of the list is called first.

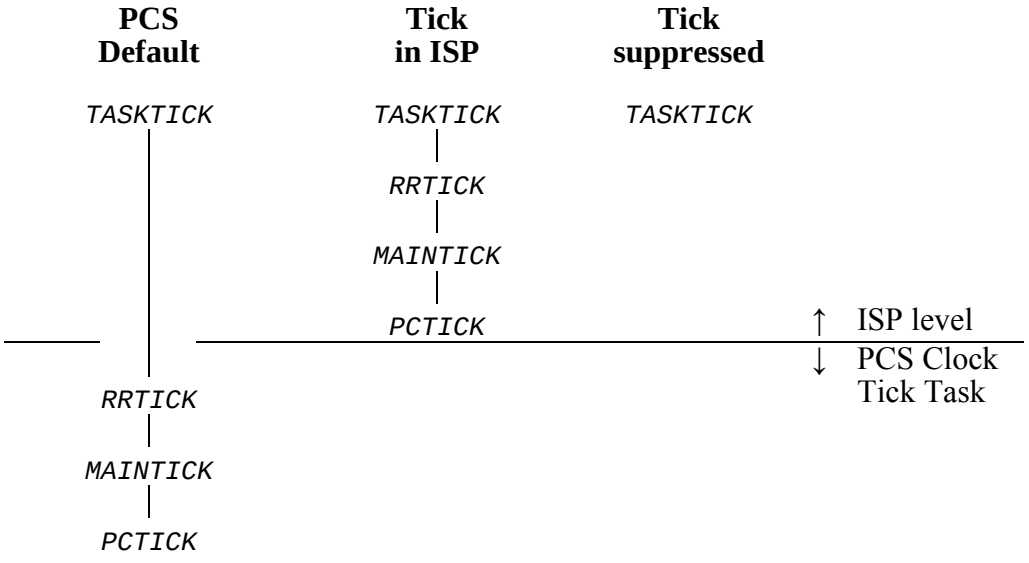

Figure B1.7-1 Software Clock Tick Sequence

Hardware Clock Hook

The PC Supervisor permits few options for clock hardware interrupt service. A hardware clock hook is provided to permit you to install a procedure which will be called by the PCS clock driver at the 18.2 Hz frequency. Your procedure must be installed by calling *ajclop* from a Restart Procedure.

The PCS hardware clock driver makes a *FAR* call to your procedure at the 18.2 Hz frequency after it has generated the software clock tick interrupt (INT 1CH). On entry to your procedure, interrupts are enabled. All registers are free for use. The procedure must end with a *FAR* return.

It is instructive to use the previous example to see the point at which your procedure is called. If your procedure is *HDWTICK*, it will be called from within the PCS clock ISP as indicated in Figure B1.7-2.

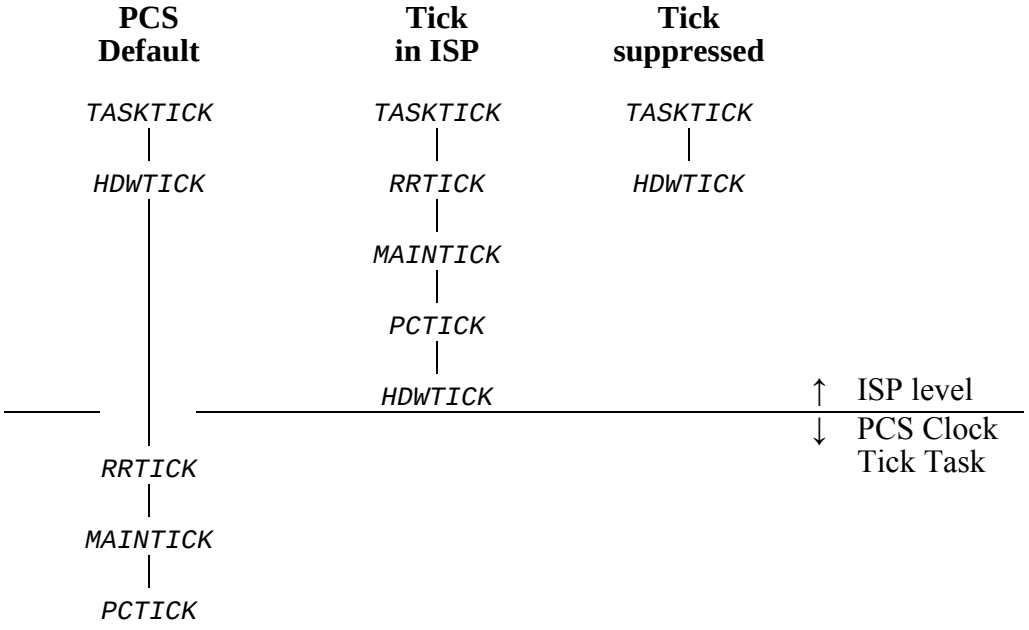

Figure B1.7-2 Hardware ISP Clock Hook Insertion

Hardware Clock Override

One final solution to the clock conundrum is to replace the PCS hardware clock driver with your own. To do this and still retain the benefits of the PC Supervisor, you must observe the following constraints.

The PC Supervisor Clock Tick Task must still be included in your AMX System Configuration Module in order to use the PCS clock, keyboard and DOS services.

You MUST use the 18.2 Hz hardware clock frequency. This means you will be forced to have an AMX tick interval which is a multiple of the imprecise 54.9 ms clock period.

Your main program must read the initial hardware interrupt vector and save the pointer to the original handler. This must be done before you launch AMX. You cannot use AMX procedure *ajivtr* to read the vector. You must not install your own handler yet.

You must provide a Restart Procedure which calls *ajclop* to indicate that the PCS hardware clock driver is to be inhibited. You must also install a new hardware clock ISP similar to that illustrated below.

```
CLKISP   PROC  FAR
         CALL  AAINT                          ;begin interrupt
         PUSHF                                ;prepare for IRET
         CALL  <FAR ptr to saved handler>     ;service PC clock
         CALL  AACLK                          ;service AMX clock
         CALL  AAINX                          ;end of interrupt
         IRET
;
CLKISP   ENDP
```

When you override the hardware clock driver in this fashion, several restrictions are imposed. The 18.2 Hz clock frequency must be used. The hardware clock hook provided by *ajclop* is not supported. You can call your hook from your clock ISP. The software clock interrupt call sequence will be as described in Figure B1.7-1.

Clock Option Procedure *ajclop*

The PC Supervisor provides procedure *ajclop* (*AACLOP*) to permit you to alter the clock options offered by the PCS clock driver.

The calling sequence, using standard AMX nomenclature, is as follows:

Setup

```
#define KOPNOT 1                /* suppress sfw tick        */
#define KOPTIC 2                /* sfw tick in ISP          */
#define KOPHDW 4                /* hardware override        */

void ispproc();                 /* ISP hdw hook             */
unsigned int options;

ajclop(options, ispproc);
       DX       ES:BX
```

Where *options* is the sum of one or more of the following clock option masks (see Figures B1.7-1 and B1.7-2).

KOPNOT Suppress software clock interrupt (INT 1CH) service.

KOPTIC Service the entire software clock interrupt chain while in the clock ISP. Do not defer service to the PCS Clock Tick Task. This option is ignored if option *KOPNOT* is specified.

KOPHDW Suppress installation of the PCS clock hardware ISP. You must provide your own clock driver as described under "Hardware Clock Override". This option is not affected by options *KOPNOT* or *KOPTIC*.

ispproc is a *FAR* pointer to a procedure to be called by the PCS hardware clock driver as described under "Hardware Clock Hook".

If you do not wish to install a hardware clock hook, set this parameter to *NULL* (*0L*).

This parameter is ignored if option *KOPHDW* is selected.

This page left blank intentionally.

AMX™ 86 Tool Guide (Appendix B)

Application Note 2

This technical note covers the following topics:

- AMX Message Length
- AMX Shutdown
- PC Clock Frequency
- PC Mice Interference
- PC Print Screen Operation
- Trouble Shooting With InSight

B2.1 AMX Message Length

AMX messages originate as user defined blocks of *AMXMSZ* (12) or more sequential bytes of memory. The maximum length (*AMXMSZ >= 12*) is determined by you when you create your System Configuration Module.

Whenever you send a message to a task or to a message exchange, you point to memory containing the message. AMX copies all *AMXMSZ* bytes into an AMX message envelope and attaches the envelope to the appropriate mailbox.

When you get a message using *ajgmsg* or *ajmxget*, AMX removes the envelope from the mailbox and copies all *AMXMSZ* bytes from the envelope to the storage area which you must provide. Failure to provide at least *AMXMSZ* bytes of alterable storage for the message is a common fault. Remember that AMX will copy *AMXMSZ* bytes.

Hint: Align messages on double word boundaries to improve execution speed and to ease porting your application to 32-bit processors.

B2.2 AMX Shutdown

If you use the AMX function *ajexit* 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. The PC Supervisor, if included in your system, is also available for use.

Once all of your Exit Procedures have been executed, AMX shuts down the PC Supervisor 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.

B2.3 PC Clock Frequency

Some PC Supervisor users have experienced significant drift in the AMX calendar clock over extended periods of time. The problem stems from the accuracy and stability of the frequency of the crystal used as input to the 8253 Programmable Interval Timer from which the PC Supervisor derives the AMX clock tick.

We have observed variations in PC clock frequency leading to gains (or losses) of as much as 10 seconds per day. These observations were made on various PCs running DOS 3.3 without AMX.

Applications requiring highly stable long term calendar clocks must not depend on the interval timer provided on most commercial PCs.

Observation: DOS users should be aware that the DOS clock will lose days if no DOS activity occurs for periods in excess of 24 hours. Try leaving your PC powered on but untouched over a weekend and then use the `DATE` command to request the date. Be sure to remove all TSRs since their activity can hide the problem.

B2.4 PC Mice Interference

Some PC mice have been observed to operate at sample rates high enough to interfere with the PC Supervisor's clock and keyboard service. This problem is most pronounced when operating in PCs for which the PC Supervisor rotates the 8259 interrupt priorities as described in Chapter 5 of the PC Supervisor Reference Manual.

B2.5 PC Print Screen Operation

The PC Supervisor Keyboard Task supports the PC print screen operation initiated by the PrtSc key. However, by default, the print screen feature is disabled.

To enable the print screen feature, a task must call PCS function *ajkbon* with a mask value of 80H. Without this call, print screen will remain inhibited.

By default, the PC Supervisor will call the print screen handler that was provided in software interrupt vector 5 at the time AMX was launched. Normally that is the ROM BIOS print screen handler. If you wish to provide your own handler, your *main* program must install a pointer to your handler into software vector 5 prior to launching AMX. It is your responsibility to save and restore the original vector content.

Once AMX is operational, software interrupt vector 5 is used by AMX for the array bounds error task trap. The vector must not be altered by your application.

PrtSc Vector Relocation

If you wish, you can relocate the print screen interrupt vector from vector 5 to another software vector of your choice.

Edit each of the Hardware Specification Blocks in PC Supervisor file *AA832PCH.ASM* to reflect your interrupt choice. Assemble the file and include it in your link specification prior to the PC Supervisor library.

Before a task calls PCS function *ajkbon* to enable print screen service, be sure to install a pointer to your print screen handler into the software interrupt vector you have chosen. The installation can be done by your *main* program, a Restart Procedure or a task.

Print Screen Handler

Your print screen handler must preserve all registers and flags. It is called with interrupts disabled and all registers in an undefined state. The direction flag is set to forward. Your handler executes in the context of the PCS Keyboard Task using its stack. The handler must end with a *FAR* return or an *IRET* instruction.

If your handler is coded in C, it should begin with a call to *ajmodl* to properly initialize the *DS* register in case you need access to C variables in group *DGROUP*.

Remember that your handler executes at the priority of the PCS Keyboard Task. It competes with all other tasks for the use of the PC screen and printer.

B2.6 Trouble Shooting With InSight

Cannot Invoke InSight in Remote Mode

The most probable problem is cabling. Read and reread Chapters 5.1, 8.3 and 8.4 of the InSight Reference Manual.

If you are using signalling, use a breakout box to look at the state of the modem control line (*DTR* or *RTS*) from your remote system to your host PC. If the line is high (+v) before you start your system (as it is when using Borland's Turbo Debugger), you will have to use inverted signal logic (see the Borland (toolset *TC*) chapter of this tool guide).

Cannot Invoke InSight from Debugger

Be sure you have the C Language option selected. If assembler is selected, the debugger will not accept the C syntax *djinsight()*. Your debugger may want you to include a leading underscore as in *_djinsight()*

It is also possible that the function *djinsight()* has not been included in your link. Look in your link map or in the debugger's symbol list. If symbol *djinsight* or *_djinsight* does not exist, you probably forgot to include object module *DJ839DBG.OBJ* in your link.

Video Problems

If your PC video interface is in a graphic mode when InSight is invoked, InSight will switch the interface to text mode. InSight always reads the current video mode before it switches to text mode. If the mode information provided by the ROM BIOS *INT 10H* function call does not define the actual video controller state, InSight will be unable to restore the video screen on exit.

This condition will occur if the ROM BIOS has been bypassed by your screen application or device driver.

Mice Problems

The resident InSight Manager on a host PC has been observed to disable a bus mouse when invoked from a remote system under test.

The hardware mouse DOS device driver was hooked into the ROM BIOS *INT 10H* video function call and was disabling the mouse whenever a video mode change request occurred.

> Note
>
> Effective April 2, 2001, InSight is no longer supported.

AMX™ 86 Tool Guide (Appendix B)

Application Note 3

This technical note covers the following topics:

- InSight Stack Checks
- Unexpected Exit from a Breakpoint
- Tool Set Caveats
- Cautions When Using DOS

B3.1 InSight Stack Checks

InSight will show task stack overflow and underflow under most circumstances. However, the detection method is not foolproof.

AMX inserts a unique pattern as a fence at the top and bottom of a task's stack when the task is created. InSight reports if either of these fences has been corrupted.

Stack overflow (stack goes below bottom of stack) can occur without the lower fence being altered. Consider a task which is near the bottom of its stack. If the task calls a procedure which allocates a large local array on the stack, it is possible that the array may actually straddle the task's lower bound. In this case, stack overflow has occurred but the fence has not been altered. The fence will remain unaltered if the array goes unused or only partially used by the procedure.

B3.2 Unexpected Exit from a Breakpoint

Have you ever been stopped at breakpoint in a task and suddenly had the rest of your AMX system resume execution? Here is how it can happen even with the AMX Breakpoint Manager in use.

When you stop at a breakpoint, the Breakpoint Manager halts all task activity so that you can examine the system state at your leisure. It begins a monitoring process to determine when you have left the breakpoint so that it can resume task activity. One of the ways it does this is by monitoring clock ticks.

Now suppose that you ask your debugger to evaluate a function `xyz()` in your target system. Some debuggers do this by restoring your program's stack and registers and then executing the function `xyz()`. Since interrupts are enabled and your task's stack is once again in use, the Breakpoint Manager concludes that you have left the breakpoint and it erroneously unleashes task activity.

This effect was first observed when the InSight function `djinsight()` was evaluated. InSight has been made breakpoint aware, provided that the Breakpoint Manager is used. InSight assures that this unexpected breakpoint exit will not occur.

B3.3 Tool Set Caveats

This AMX 86 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 tool sets. In fact, from time to time, the information in this Tool Guide will be superceded by newer releases of the tools from the tool vendors.

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, Borland's linker cannot necessarily link object modules produced by the Microsoft C compiler.

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 module using a new tool, it is advisable to rebuild all AMX modules with the new tool. Follow the guidelines provided in Appendix A.

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.

B3.4 Cautions When Using DOS

When using DOS 5.0 and later, the following cautions and restrictions must be observed.

Do not use the DOS command *SETVER* to change the DOS version to any value less than 3.1.

DOS function *4B05H* to set the execution state of a new program is NOT supported.

Do NOT put the AMX DOS Device Driver *AA832PDD.SYS* in high memory.

Significant memory saving and speed improvement for interrupt service can be achieved by including the statement

```
STACKS=0,0
```

in your *CONFIG.SYS* file. Since the AMX Interrupt Supervisor switches to the private AMX Interrupt Stack to service interrupts, the unnecessary overhead of DOS interrupt stack switching is eliminated.

Finally, recall that all of the following DOS features are shared globally by your AMX tasks and the PCS DOS Command Task. Therefore you must use these features with caution treating the features as shared resources.

All country extensions
All code page extensions
All *IOCTRL* extensions
Upper memory link (DOS 5.0 functions *5802H* and *5803H*)

This page left blank intentionally.

AMX™ 86 Tool Guide (Appendix B)

Application Note 4

This technical note provides information and guidance concerning your use of AMX 86. The topics covered are:

- Stack and Data Alignment
- AMX Message Caveats
- Spurious Interrupts and the 8259 PIC
- Make File for AMX Applications
- C Floating Point Operations
- Using 32-bit Registers with AMX 86
- AMX 86 Interrupt Latency

B4.1 Stack and Data Alignment

AMX 86 is delivered to you ready for use on any Intel 80x86 compatible processor. 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 32-bit 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 32-bit aligned to begin with.

Typically it is your program linker which ensures proper alignment. For example, your linker may provide a directive to force specific segments (code, data, etc.) from each module in your link to be 32-bit aligned. All data and stacks in the AMX System Configuration Module will then be properly aligned.

Failure to properly align AMX data segments will almost certainly lead to performance degradation.

B4.2 AMX Message Caveats

An AMX message consists of *AMXMSZ* sequential bytes which are passed by value to a task through a task's mailboxes or through a message exchange. The sender may be a task, ISP, Timer Procedure, Restart Procedure or Exit Procedure.

AMXMSZ is 12 by default but can be increased by you in your System Configuration Module. AMX delivers the message by copying *AMXMSZ/4* 32-bit values to the message destination.

On processors such as the 80386, the message copy may be slow if the message source (or destination) is not long word aligned. The following examples illustrate BAD coding techniques which may lead to slow execution at runtime.

```
extern AMXID taskid;

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

void badcode(void) {
   ajsend(taskid, 0, messageA);
   ajsend(taskid, 0, &messageB);
   }
```

The first message, *messageA*, is a constant character array which most C compilers will place in memory at a long word 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 word aligned, the AMX procedure *ajsend* may execute slowly because of the long word 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 word 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 word 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 of *AMXMSZ=12*. Only the first *AMXMSZ* characters, i.e. *"Fixed long m"*, will be sent in the AMX message. Also note that no trailing null character *'\0'* will be present in the AMX message.

B4.3 Spurious Interrupts and the 8259 PIC

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

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

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

> PC Supervisor Note
>
> If you use the AMX 86 PC Supervisor, you will only have to consider spurious interrupts if your application services the IRQ7 interrupt on either of the two PC/AT 8259 interrupt controllers.

Example: Application Ignores IRQ7

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

To create default spurious interrupt handlers for both the master and slave controllers, insert the following code fragment into one of your assembly language modules.

```
IRQ_CODE SEGMENT BYTE 'CODE'
          ASSUME CS:IRQ_CODE, DS:NOTHING, ES:NOTHING, SS:NOTHING
;
          PUBLIC _ATMASTER          ; Master 8259 ISP
          PUBLIC _ATSLAVE           ; Slave 8259 ISP
;
; Master & Slave Spurious IRQ7 Interrupt Service Procedures
; Interrupts are disabled
; Clear spurious interrupt on slave 8259 with a
; non-specific EOI to the master 8259
;
_ATSLAVE  PROC    FAR
          PUSH    AX
          MOV     AL,020H
          OUT     20H,AL            ; n/s EOI to master 8259
          POP     AX
_ATMASTER LABEL   FAR
          IRET
;
_ATSLAVE  ENDP
;
IRQ_CODE  ENDS
```

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

```
#include "amx831cf.h"

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

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

void cdecl rr8259(void) {
                                    /* Install Master 8259 IRQ7 ISP   */
   ajivtw(M8259BASE + 7, (void (*)())ATMaster);

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

Note that the Restart Procedure in this example unconditionally installs the ISP pointers into the Interrupt Vector Table (IVT) as is appropriate for an AMX application launched for permanent execution. If your AMX application is launched for temporary execution, you should save the initial values of each IVT entry using *ajivtr*. Then create an Exit Procedure which uses *ajivtw* to restore the IVT entries to their initial values when your AMX application shuts down.

Example: Application Uses IRQ7

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

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

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

```
#include "amx831sd.h"
#define M8259 0x20                  /* Master 8259 device address     */
#define M8259BASE 0x08              /* Master 8259 IRQ0 base          */

void cdecl IRQ7isp(void);           /* ISP for IRQ7                   */
static struct amxisps IRQ7root;     /* ISP root for IRQ7 ISP          */
static int in_service;              /* Private boolean                */

void cdecl IRQ7rr(void) {           /* IRQ7 Restart Procedure         */
   in_service = 0;                  /* IRQ7 is not in service         */
   ajispm(IRQ7isp, &IRQ7root);      /* Make/install IRQ7 ISP root     */
   ajivtw(M8259BASE + 7, (void (*)())&IRQ7root);
   }


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

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

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

B4.4 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.

AMX Header Dependencies

Do not blindly list the AMX header files included in your C source files as the only AMX dependencies in your make specification.

AMX header file *AMX831SD.H* unconditionally includes file *AMX831CF.H*. Hence, any file which depends on *AMX831SD.H* also depends on *AMX831CF.H*.

In other cases, AMX header file *AMX831CF.H* will conditionally include file *AMX831SD.H*. Hence, any file which depends on *AMX831CF.H* may also depend on *AMX831SD.H*.

System Configuration Module

Your AMX System Configuration Module *SYSCFG.ASM* 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.ASM*.

Declare your System Configuration Module *SYSCFG.OBJ* to depend on the User Parameter File *SYSCFG.UP*. The make directive to create *SYSCFG.OBJ* consists of two statements, one to make the source file *SYSCFG.ASM* and one to assemble it.

The System Configuration Module source file *SYSCFG.ASM* is generated using the AMX Configuration Generator as follows:

```
AM831CG SYSCFG.UP AM831CG.CT SYSCFG.ASM
```

The Configuration Generator *AM831CG.EXE* combines the information from your User Parameter File *SYSCFG.UP* with the AMX System Configuration Template File *AM831CG.CT* to produce file *SYSCFG.ASM*. The source module must then be assembled as described in the chapter of the AMX Tool Guide for the toolset which you are using.

B4.5 C Floating Point Operations

C compilers provide floating point support in their runtime libraries. However, most implementations are not reentrant. Libraries which support math coprocessors are not reentrant because the math chip internal stack and registers contain the floating operands and results. Emulation libraries are usually not reentrant because they use global, and sometimes static (i.e. hidden), variables to hold the floating operands and results.

Identifying C functions that use floating point operations is not always easy. For example, a C function that receives a pointer to a structure containing floating point operands can operate on those operands using the +, -, * and / operators, none of which are easily identified as floating point operators. Hence, the C compiler can generate calls to floating point library functions making the C function non-reentrant without your even knowing it.

Single Task Access

Several solutions are possible. The simplest is to have only one task responsible for all floating point operations. The task is usually designed to receive AMX messages which identify the operation or sequence of operations to be performed.

In some cases, the AMX message provides a directive and a set of operands on which to operate. In other cases, the message might include a pointer to a function in the calling task's domain which the floating point task executes on behalf of the calling task.

An alternative is to use an AMX resource semaphore to control access to and ownership of the C runtime floating point library. Tasks wishing to use floating point must first gain ownership of the resource with a call to the AMX Semaphore Manager. The task must be sure to free the resource when it is no longer required. Of course, the disadvantage of this approach is that a low priority task can inadvertently prevent a higher priority task from using the library.

Multiple Task Access

An alternative is to use the AMX task scheduling hooks to preserve the floating point context across task switches. This sounds simple, but may not be easy to implement. For example, if an emulation library is being used, the global variables being used by the library must be saved and restored at each switch. The variables may be difficult to identify and may be subject to change with each release of new libraries by the C compiler vendor.

If a math coprocessor is used, the chip contents must be extracted and restored at each task switch. Of course you may have to delay while an operation in progress completes so that the internal register content can be extracted. Such delays may lead to a degradation in AMX task switching performance.

The 16 byte region of the Task Control Block (TCB) reserved for your use can be used to advantage. The region is zeroed by AMX when the task is created. If a task uses floating point, it can install a pointer to a storage block for use by your Scheduler Hooks. The hook functions can fetch the pointer from the TCB. If the pointer exists, the hook function must preserve or restore the floating point context in the task's private storage.

B4.6 Using 32-bit Registers with AMX 86

AMX 86 is frequently used on 32 bit Intel386™, Intel486™ or Pentium™ processors, albeit in real mode. The question arises: can 32-bit instructions and registers be used?

The 32-bit processors allow many 32-bit instructions to be executed while in real mode. Execution of such instructions does not affect the proper operation of AMX.

However, using instructions which manipulate the contents of the 32-bit registers may lead to difficulty.

Simply stated, AMX 86 is a 16-bit product. Only 16-bit registers are saved across task context switches. The `FS` and `GS` segment registers are not saved at all. Consequently, the upper half of 32-bit registers and the `FS` and `GS` registers can get lost across task switches.

If you must use 32-bit registers, do it in a single task. Alternatively, use an AMX resource semaphore to control single task access to and use of the registers.

If a task is allowed to use the `FS`, `GS` or 32-bit registers, do not allow interrupt handlers to touch them. Interrupt handlers can only use the registers if tasks do not.

B4.7 AMX 86 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. Do not use AMX Scheduler Hooks (Chapter 13.3 in the AMX 86 User's Guide). The worst case AMX interrupt latency occurs in the path through your hooks into the AMX scheduler.

Appendix C. AMX 86 Device Drivers

C1. AMX 86 Clock Drivers

C1.1 Clock Driver Operation

You must provide a clock driver as part of your AMX application so that AMX can provide timing services. AMX clock drivers are provided with AMX for the timer chips used on the boards with which AMX has been tested. These drivers are ready for use and can be installed as described in Chapter 5.2 of the AMX 86 User's Guide. The clock drivers are delivered in **chip support** source files having names of the form `CHnnnnT.C` where *nnnn* identifies the particular clock chip. The clock chip support procedures are named `chxxxxxxx`. The files can be found in installation directory `AMX831\SAMPLE`.

An AMX clock driver consists of three parts: an initialization procedure, a clock Interrupt Service Procedure (ISP) and an optional shutdown procedure.

Clock Startup

The **clock initialization procedure** must configure the real-time clock to operate at the frequency defined in your AMX System Configuration Module. It can then install the pointer to the clock ISP root into the Interrupt Vector Table and start the clock.

Care must be taken to ensure that clock interrupts do not occur until the clock is properly configured and the pointer to the clock ISP root is present in the Interrupt Vector Table.

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

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

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

Clock Interrupts

A real-time clock used with the 80x86 processor will interrupt through an entry in the Interrupt Vector Table. The processor will automatically dispatch to your clock ISP. Your clock ISP must be built as described in Chapter 5.2 of the AMX 86 User's Guide.

For fastest service, the **clock ISP** can be coded in assembly language. The clock ISP dismisses the clock interrupt request and calls the AMX Clock Handler function *AACLK* to ensure that the clock interrupt is recognized by AMX as the source of its fundamental clock tick operating at the frequency and resolution defined in your AMX System Configuration Module.

A clock ISP coded in C consists of an ISP root and an Interrupt Handler. The processor dispatches to the ISP root in response to the clock interrupt request. The ISP root calls the clock Interrupt Handler to dismiss the clock interrupt request. The clock Interrupt Handler must call the AMX Clock Handler function *ajclk()* to inform AMX that a hardware clock tick has occurred.

Clock Shutdown

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

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

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

PC Supervisor Note

The PC Supervisor includes its own 8253 clock driver for the PC/AT. If you use the PC Supervisor, do NOT include your own clock ISP in your AMX system.

C1.2 Custom Clock Driver

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

The I/O port address of the clock is at *0x03C0*.
The clock interrupt is generated using vector number 125.
The clock interrupt is dismissed by writing bit pattern *0x08* to the clock register at its device address plus 4.
An interrupt controller is not used.

An assembly language clock ISP for such a device could be coded as follows:

```
            PUBLIC   _clockisp
_clockisp   PROC     FAR
            PUSH     DX              ; Save some registers
            PUSH     AX
            MOV      DX,03C4H        ; DX = clock device address + 4
            MOV      AL,8            ; AL = 8
            OUT      DX,AL           ; Dismiss interrupt
            POP      AX              ; Restore registers
            POP      DX
            CALL     AACLK           ; AMX Clock Handler
            IRET                     ; Return from interrupt
```

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

```
void cdecl clockisp(void);                /* External clock ISP       */

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

   /* Install pointer to clock ISP into Interrupt Vector Table        */
   ajivtw(125, (void (*)())clockisp);

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

C1.3 Intel 8253 (8254) Clock Driver

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

The 8253 timer generates IRQ0 on the PC/AT master 8259 interrupt controller which is assumed to use the block of 8 vectors at vector *0x08* in the Interrupt Vector Table.

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

Clock driver module *CH8253T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the PC/AT 8253 (8254) Clock Driver

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

C1.4 Am186ES Clock Driver

The AMX clock driver for the AMD Am186ES embedded timer controller is ready for use on the AMD Net186 Evaluation Board or on the VAutomation iCON186 TIPS3 Evaluation Board. It is configured to use timer channel 1 operating at 1 KHz (1 ms period). **Source code** for this AMX clock driver is provided in file *SAMPLE\CH186EST.C*.

The Am186ES generates its timer 1 interrupt using vector *0x12* in the Interrupt Vector Table.

You must compile clock source module *CH186EST.C* and link the resulting object module with the rest of your AMX application.

Clock driver module *CH186EST.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the Am186ES Clock Driver

If you wish to use a different Am186ES timer channel or change the timer frequency, you must edit the definitions in source file *CH186EST.C* and recompile the module. Edit instructions are included in the file.

This clock driver can be easily adapted for use with any embedded controller which incorporates peripheral devices compatible with the Intel 80186 processor architecture.

C1.5 PC/AT Clock Driver

The AMX PC/AT clock driver is ready for use on any conventional PC/AT computer system. It intercepts and uses the PC/AT timer without any modification to its method of operation. **Source code** for this AMX clock driver is provided in file *SAMPLE\CH_PC_T.C*.

The timer generates IRQ0 on the PC/AT master 8259 interrupt controller which is assumed to use the block of 8 vectors at vector *0x08* in the Interrupt Vector Table. This clock driver does not alter the PC/AT timer hardware settings. The timer continues to operate at the frequency dictated by your PC/AT hardware configuration.

You must compile clock source module *CH_PC_T.C* and link the resulting object module with the rest of your AMX application.

Clock driver module *CH_PC_T.C* includes the clock initialization procedure *chclockinit* and the clock shutdown procedure *chclockexit*. Insert procedure *chclockinit* into the list of Restart Procedures provided in your System Configuration Module at the point at which you wish the clock to be enabled during the launch. If you intend to launch AMX for temporary execution, insert *chclockexit* into the list of Exit Procedures at the point at which you wish the clock to be disabled during the shutdown.

Porting the PC/AT Clock Driver

This clock driver is only suitable for use on a conventional PC/AT computer system.

If your PC/AT uses a different timer channel or uses a different interrupt vector number, you must edit the definitions in source file *CH_PC_T.C* and recompile the module. Edit instructions are included in the file. Note that you will have to add support for interrupt operation via a slave 8259 interrupt controller if so required.

C2. AMX 86 Serial Drivers

C2.1 Serial Driver Operation

The AMX Sample Program uses a simple serial driver to access a terminal device for keyboard input and display output. This serial driver supports an asynchronous I/O interface device, commonly called a UART, connected by a 3-wire cable to the terminal. 8-bit characters are transmitted and received as serial bit streams framed according to the manner in which the UART is configured. The serial drivers initialize the UART for 8-bits of data, no parity bit, one start bit and one stop bit. The UART is set for 9600 baud operation.

AMX serial drivers are provided with AMX for the serial I/O interfaces used on the boards with which AMX has been tested. These drivers are ready for use with the AMX Sample Program. The serial drivers are delivered in **chip support** source files having names of the form *CHnnnnS.C* where *nnnn* identifies the particular serial chip. The files can be found in installation directory *AMX831\SAMPLE*.

Application Interface Procedure *chuart()*

Application interface procedure *chuart()* provides services to initialize the UART, sense its status, and read and write 8-bit characters.

The serial driver supports two **logical ports**. Logical port 0 is assumed to be used by your remote debugger. Logical port 1 is reserved for your application. Serial device drivers which support more than one serial device interface translate these logical port designations into physical equivalents.

The setup and calling sequence is as follows:

```
/* Define op codes                                                  */
#define CONCFGR 0           /* Initialize port (param = baud rate)  */
#define CONSTAT 1           /* Read status register (param unused)  */
#define CONREAD 2           /* Read from data register (param unused)*/
#define CONWRITE 3          /* Write to data register (param = char)*/

/* Define returned status bit masks                                 */
#define CONRRDY (0x01)      /* Read ready (character available)     */
#define CONWRDY (0x20)      /* Write ready (character can be sent)  */


int cdecl chuart(           /* Function prototype                   */
      int port,             /* Logical port                         */
      int opcode,           /* Opcode (see definitions)             */
      long param);          /* Operation parameter                  */
```

The operation specified by parameter *opcode* is performed using the logical port indicated by *port*. If parameter *param* is not required for the operation, set it to *0L*.

The value *-1* is returned if the logical port is invalid or not supported by the serial driver.

When configuring the UART (*opcode* = *CONCFGR*), parameter *param* is the baud rate to be used. For example, set param to *19200L* to operate at 19200 baud. If *param* is *0L*, the default value of 9600 baud is used. The value *0* is returned once the initialization is complete.

When sensing status (*opcode* = *CONSTAT*), parameter *param* is ignored. The returned value (call it *status*) indicates the UART transmit and receive status. A character is present in the receive register, ready to be read, if *status&CONRRDY* is non-zero. The transmitter register is free for use, ready to be accept a character for transmission, if *status&CONWRDY* is non-zero.

When fetching a character (*opcode* = *CONREAD*), parameter *param* is ignored. The returned value is the character read from the device, cast to be an *int* value and masked to 8-bits. The function waits for a character and does not return until one is available.

When transmitting a character (*opcode* = *CONWRITE*), parameter *param* is the 8-bit character, cast to be a *long* value. The function does not return until the character has been written into the transmit register. The value *0* is returned once transmission has been initiated.

C2.2 Intel 8250 Serial Driver

The AMX serial driver for the Intel 8250 UART is ready for use on either a PC/AT or on hardware which incorporates the Intel386EX processor. It is configured to support two UARTs: logical port 0 at port address 0x03F8 and logical port 1 at port address 0x02F8. The UART input clock frequency is defined to be 1.842 MHz. **Source code** for this AMX serial driver is provided in file *SAMPLE\CH8250S.C*.

If your PC/AT uses different UART addresses or a different UART clock frequency or if you wish to change the logical port assignments, you must edit the definitions in source file *CH8250S.C*. Edit instructions are included in the file.

You must compile serial driver source module *CH8250S.C* and link the resulting object module with the rest of your AMX application.

C2.3 Am186ES Serial Driver

The AMX serial driver for the AMD Am186ES embedded UART controller is ready for use on the AMD Net186 Evaluation Board or on the VAutomation iCON186 TIPS3 Evaluation Board. It is configured to support two UARTs: logical port 0 at port address 0xFF10 and logical port 1 at port address 0xFF80. The UART input clock frequency is defined to be 24.0 MHz as required for the VAutomation board. **Source code** for this AMX serial driver is provided in file *SAMPLE\CH186ESS.C*.

If your board uses different UART addresses or a different UART clock frequency or if you wish to change the logical port assignments, you must edit the definitions in source file *CH186ESS.C*. Edit instructions are included in the file. To use this serial driver on the AMD Net186 Evaluation Board, you must change the input clock frequency definition to be 40.0 MHz.

You must compile serial driver source module *CH186ESS.C* and link the resulting object module with the rest of your AMX application.

C2.4 PC/AT Console Driver

The AMX PC/AT console driver is ready for use on any conventional PC/AT computer system. It is configured to use the PC/AT keyboard and display as a local console device. **Source code** for this AMX console driver is provided in file *SAMPLE\CH_PCCON.C*.

The console driver uses C runtime library functions *_kphit()*, *_getch()* and *_putch()* for console I/O operations. These functions (or equivalents) are recognized by the console driver for Borland (*TC*), Microsoft (*MC*) and WATCOM (*WC*) toolsets. Since the Paradigm tools do not provide these C functions, the console driver uses a BIOS call for keyboard input and does direct video memory writes for screen display when used with the AMX 86 *PD* or *PX* toolset.

The logical port number and the UART initialization op code *CONCFGR* are ignored by the console driver's *chuart()* interface function.

When op code *CONSTAT* is used to sense console status, mask bit *CONWRDY* in the returned status value is always set, indicating that the console is always ready to accept a character for display.

You must compile console driver source module *CH_PCCON.C* and link the resulting object module with the rest of your AMX application.

Appendix D. AMX 86 Timing Guide and Data

The AMX 86 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 86 Timing Guide, although included as Appendix D of this AMX 86 Tool Guide, is constructed as a separate "manual" and is page numbered accordingly. Hence, the AMX 86 Timing Guide can be viewed as a self-contained document, separate from the AMX 86 Tool Guide in which it resides.

This page left blank intentionally.

Appendix E. AMX 86 ROM Option

An AMX system can be configured in two ways. The particular configuration is chosen to best meet your application needs.

Most AMX systems are linked. Your AMX application is linked with your System Configuration Module and the AMX Library. The resulting load module is then copied to memory for execution either by loading the image into RAM or by committing the image to ROM. Such a ROM contains an image of your application merged with AMX in an inseparable fashion.

The AMX ROM option offers an alternate method of committing AMX to ROM. The ROM option allows the subset of AMX and its managers required by your application to be linked together without any application code to form a separate AMX ROM image. The resulting ROM can be located anywhere in your memory configuration. The penalty paid for ROMing in this fashion is slightly slower access by application code to AMX services.

> PC Supervisor Note
>
> If you use the AMX PC Supervisor, you CANNOT use the AMX 86 ROM Option.

Selecting AMX ROM Options

To support an AMX ROM system, the following files are provided.

AMX831RO.DEF	AMX ROM Option Definitions
AA831ROP.ASM	AMX ROM Option Module
AA831ROS.ASM	AMX ROM Option Entry Module
AA831ROS.OBJ	
AA831RAC.ASM	AMX ROM Access Module

The AMX ROM option definitions must be edited to select the particular AMX managers which you wish to include in your ROM. Copy the file *AMX831RO.DEF* and edit the sequence of constant declarations as explained at the beginning of the file.

By default, file *AMX831RO.DEF* includes all AMX managers in the AMX ROM.

AMX requires its own private data segment. The location of this data segment must be defined by you when you edit file *AMX831RO.DEF*. You must specify the absolute memory address of a paragraph aligned block of RAM memory reserved for use by AMX. For 24-bit memory systems, the memory must be page aligned.

There must be sufficient memory at the location specified for the AMX Data Segment to meet the worst case needs of any AMX system which uses your AMX ROM.

You must edit the AMX ROM option definitions file *AMX831RO.DEF* to define the location of your AMX ROM. You must specify the absolute memory address of a paragraph aligned block of ROM memory reserved for AMX code. For 24-bit memory systems, the memory must be page aligned.

By default, the AMX ROM is assumed to be located at address *7000:0* (absolute address *70000H*). AMX RAM is assumed to be at address *8000:0* (absolute address *80000H*).

Edit your copy of file *AMX831RO.DEF* to reflect your choice of run-time ROM and RAM locations.

Creating an AMX ROM

Once the option definitions have been edited, use your assembler to assemble the AMX ROM option module *AA831ROP.ASM* to produce object file *AA831ROP.OBJ*. Your AMX ROM image can then be created by linking this module with the ROM entry module and the AMX Library. Follow the instructions provided in the toolset dependent chapters of this manual for linking a separate AMX ROM.

Note that the AMX ROM Option Entry Module *AA831ROS.OBJ* must be linked with your ROM Option Module *AA831ROP.OBJ* to avoid link errors by some linkers. The entry module defines a dummy start address for the AMX ROM to keep the linkers happy.

The resulting AMX ROM image file is then committed to ROM using conventional ROM burning tools. The manner in which this is accomplished will depend completely upon your development environment. In general, the process involves the conversion of the AMX ROM image file to a hex file for transfer to a PROM programmer.

As an example, your AMX ROM image can be created using the Microsoft assembler *MASM* and linker *LINK* as follows.

```
MASM AA831ROP /ML /N,,;
LINK AA831ROP+AA831ROS,AMXROM,AMXROM,AMX831.LIB;
```

In this example, the linker will produce an executable file *AMXROM.EXE* and a map file *AMXROM.MAP*. The executable file *AMXROM.EXE* must be converted to a hex file and transferred to ROM.

Linking for AMX ROM Access

Access to the AMX ROM is via the ROM access module *AA831RAC*.

Once the option definitions have been edited, use your assembler to assemble the AMX ROM access module *AA831RAC.ASM* to produce object file *AA831RAC.OBJ*. The module is assembled in exactly the same manner as your System Configuration Module *SYSCFG.ASM* according to the directions in the toolset dependent chapters of this manual.

The AMX ROM access module provides access to all of the procedures of AMX and the subset of AMX managers which you included in your ROM. These ROM access procedures make software jumps to the ROM resident procedures.

To create an AMX system which uses your AMX ROM, proceed just as though you were going to include AMX as part of a linked system. Your System Configuration Module must indicate that AMX and its managers are in a separate ROM. To meet this requirement, you may have to use the AMX Configuration Manager to edit your User Parameter File accordingly and regenerate your System Configuration Module. If you do so, do not forget to reassemble the System Configuration Module.

Your AMX application is then linked as described in the toolset dependent chapters of this manual. However, since AMX and a subset of its managers are in ROM, you must include the AMX ROM Access Module *AA831RAC.OBJ* in your list of object modules to be linked. By so doing, you will preclude the inclusion of AMX and its managers from the AMX Library *AMX831.LIB*.

Note that you must still include the AMX Library *AMX831.LIB* in your link in order to have access to the small subset of AMX procedures which are never installed in your AMX ROM.

Once linked, your AMX application can be downloaded into RAM memory in your target hardware configuration. Alternatively, your application can be transferred to ROM using the same techniques that were used to produce the AMX ROM. Regardless of the manner in which your AMX system is loaded into your target hardware, access to the AMX ROM via the ROM Access Module is now possible.

For simplicity, the complexities which you will encounter when trying to commit the C Runtime Library to ROM have been ignored. Refer to your C compiler reference manual for guidance in ROMing C code and data in embedded applications.

> Warning!
>
> If your AMX ROM was created without a particular manager, then an AMX fatal exit will occur if your system attempts to access that manager.

Moving the AMX ROM

The AMX ROM is not position independent. Nor is the location of the AMX Data Segment.

To move either, you must edit the AMX ROM option definitions file *AMX831RO.DEF* to define the new location of the AMX ROM and AMX RAM. Assemble file *AA831ROP.ASM* and link a new AMX ROM image. Assemble the AMX ROM access module *AA831RAC.ASM* to produce a new object file *AA831RAC.OBJ* and link your AMX system with this new object file.