AMX™ 86 C Programming Guide

for use with the

AMX 86 Multitasking Executive

First Printing: **June 16, 1993**
Last Printing: **March 1, 2005**

Copyright © 1993 - 2005

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

TECHNICAL SUPPORT

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

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

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

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

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

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

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

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

DISCLAIMER

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

TRADEMARKS

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

AMX 86 C PROGRAMMING GUIDE

Table of Contents

Page

1. C Programming Primer **1**

1.1 C Programming Practices 1
1.2 Structure Packing 3
1.3 Reentrancy and Concurrent Execution 4
1.4 Using the C Runtime Library 6
1.5 Toolset Caveats 8
1.6 Segmentation Considerations 9
1.7 AMX Segment Prefix Module 15

2. AMX Programming Hints **17**

2.1 C Code Portability 17
2.2 AMX Stack Allocation 18
2.3 Choosing a Synchronization Method 19
2.4 AMX Caveats 22

3. Debugging an AMX Application **25**

3.1 Breakpoints and Tracing 25
3.2 Debugging the Launch 26
3.3 Debugging Caveats 28

4. Sample AMX System **29**

This page left blank intentionally.

1. C Programming Primer

1.1 C Programming Practices

With the exception of your AMX 86 System Configuration Module, all of your AMX application modules can be coded in C. Tasks, Timer Procedures, Restart Procedures and Exit Procedures are readily coded in C. Interrupt Service Procedures can also be coded in C although assembler is recommended. Some procedures such as AMX Task Scheduler hooks must be coded in assembler.

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

It is assumed that you are thoroughly familiar with the AMX 86 User's Guide and the relevant chapter of the AMX 86 Tool Guide for the particular C compiler that you are using.

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

Procedure Prototyping

The AMX C Interface header file *AMX831CF.H* includes function prototypes for all AMX procedures. By default prototyping is enabled.

If your C compiler allows a symbol to be defined on the command line, you can readily disable function prototyping by defining symbol *AMCCFNPX*. For example, the Microsoft C Compiler uses the */D* command-line option to define symbol *AMCCFNPX* as follows:

```
C:>CL /DAMCCFNPX filename
```

When function prototyping is disabled, all AMX functions are still declared in header file *AMX831CF.H*. The declarations simply remove all formal parameter specifications.

It is recommended that function prototyping only be disabled if it is not supported by your C compiler.

AMX Typedefs

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

The AMX header file *AMX831CF.H* includes the following C *typedef* of symbol *AMXID*:

```
typedef unsigned int AMXID;
```

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

Global and Static Variables

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

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

C allows global variables to be initialized. For example:

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

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

Initialized global variables are set to their initial value prior to starting your *main* program. Uninitialized global variables are set to zero. These initializations may not take place in a ROM system. Refer to Chapter 1.4 for a more detailed discussion of C startup requirements in ROMed systems.

Far Pointers

Some AMX procedures allow the use of *FAR* pointers (16-bit offset and 16-bit selector). The definition of *FAR* in the AMX C Interface header file *AMX831CF.H* is chosen to match the keyword used by a particular C compiler (*_far, far or FAR*).

If you use the defined keyword *FAR* in your definition of *FAR* pointers, your AMX application modules will be more readily portable to different C compilers.

Unfortunately, the C compiler manufacturers cannot yet agree on the position of the keyword in your C declarations. The following standard has been adopted in this reference manual.

```
data pointer      void FAR *
code pointer      void FAR (*)()
```

Atomic Variable References

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

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

1.2 Structure Packing

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

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

For example, the following AMX message includes 10 bytes of information.

```
struct {
       char   c1;
       long   longv1;
       char   c2;
       long   longv2;
       } msg;
```

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

Most C compilers provide an option to force byte alignment of variables at the expense of marginally slower execution speed. Judicious choice of message structures can also be used to eliminate the problem. For instance, if characters *c1* and *c2* are moved to follow long *longv2* in the example, the message length reduces to 10 bytes since gaps are avoided.

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

1.3 Reentrancy and Concurrent Execution

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

Here is an example of a non-reentrant procedure:

```
/* Return x-squared + x (non-reentrant)                    */

int poly(int x)
{
   static int tmp;

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

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

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

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

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

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

1. Tasks execute concurrently with each other, with Interrupt Service Procedures (ISPs), with Timer Procedures and with Exit Procedures. Tasks do not execute concurrently with Restart Procedures.

2. Interrupt Service Procedures execute concurrently with tasks, with Timer Procedures, with Exit Procedures and, in the case of nested interrupts, with other ISPs. They may execute concurrently with Restart Procedures if you start your devices in a Restart Procedure.

3. Timer Procedures execute concurrently with tasks, with ISPs and with Exit Procedures. They do not execute concurrently with each other or with Restart Procedures.

4. Restart Procedures may execute concurrently with ISPs if you start your devices in a Restart Procedure. Restart Procedures do not execute concurrently with each other or with any other procedure.

5. Exit Procedures execute concurrently with tasks, with Timer Procedures and with ISPs. They do not execute concurrently with each other or with Restart Procedures.

1.4 Using the C Runtime Library

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

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

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

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

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

Memory Management and *malloc*

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

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

Use the AMX Memory Manager if you require dynamic memory allocation. Have your *main* program call *malloc* to allocate the largest available region of memory possible. Save the pointer to this region for use as an AMX memory section. Then, in your AMX Memory Assignment Procedure, give the memory section to the AMX Memory Manager. Tasks can then dynamically allocate memory from this memory section.

Using Third-Party Libraries

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

C Startup Code and ROM

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

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

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

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

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

The C startup code also sets all uninitialized data to zero prior to calling your *`main`* program.

Most tool vendors include the source for the C startup code so you can change it to meet your needs. If you choose to provide your own startup code, avoid declaring a procedure called *`main`* so that the C startup code is not automatically included in your link. Your startup code must then call a C procedure with a name other than *`main`* to start your application.

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

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

1.5 Toolset Caveats

The 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 toolsets. In fact, from time to time, the information in the Tool Guide will be superceded by newer releases of the tools from the tool vendors.

KADAK tries to keep the 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 source or object modules using a new release of a tool, it is advisable to rebuild all AMX modules with the new tool. Follow the directions provided in the AMX Tool Guide.

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.

1.6 Segmentation Considerations

The Intel 8086 family of microprocessors permit access to one megabyte of memory. Memory access is controlled by the 8086 using a segmentation scheme. The method of memory access adopted by Intel for the 8086 complicates the programming of the microprocessor unless specific rules are adopted for allocating memory to program modules and controlling the access to memory by these modules.

It is the purpose of this chapter to describe the programming rules recommended by KADAK which apply to users of the AMX 86 Multitasking Executive.

It is assumed that you are modestly familiar with the 8086 segmentation architecture as described in the Intel 8086 Programmer's Reference Manual. You should also refer to the discussions of segmentation provided in the reference manuals accompanying your 8086 development tools.

Absolute Address or Physical Address

An absolute address or physical address specifies a location within the addressing range of the processor. It is the value that will be placed on the address bus when the location is to be accessed. This is a 20-bit value for the 8086 ranging from `00000H` to `FFFFFH`. Absolute addresses are almost never used in programs because the addressing methods provided by the 8086 instruction set do not support them.

Base:Offset Address

A base:offset address is the type of address used in 8086 programs. The address consists of a 16-bit base and a 16-bit offset. The 20-bit absolute address is formed from the base:offset by multiplying the base by 16 and adding the offset. Thus `2102H:0413H` would be equivalent to the absolute address `21433H`. Note that one absolute address may be equivalent to many different base:offset addresses. For example, `1A31H:7123H` is also equivalent to the absolute address `21433H`. However, in practice such equivalents do not occur because overlapped segments are not allowed by most language translators and linking loaders.

Segment

A segment is a contiguous region of memory defined by a unique base address. The maximum size of a segment is 64K bytes. The segment may be as small as desired. AMX programs will normally consist of several program segments. These segments are loaded into memory nearly adjacent to each other, often aligned to an even absolute address to improve execution speed on the 8086 (but not on the 8088).

The 8086 instruction set is tailored for segmented accessing of memory. Programs load the segment base address into a segment register and then access memory in the segment using offsets measured from the start of the segment. The processor automatically translates these base:offset addresses into absolute addresses.

The same piece of data may be accessed by using different segment bases; that is, segments may overlap. Most program language translators and linking loaders do not support segment overlapping and its use is strongly discouraged.

Pointers

A pointer is used to point to (i.e. reference) a memory location. A **small pointer** consists of only a 16-bit offset. The base address to be used in determining the memory location is assumed to be the current contents of one of the segment registers. A **large or extended pointer** consists of a 16-bit base and a 16-bit offset which are used as a base:offset address to compute the memory location.

Segment Name

A segment name is an arbitrary name used to identify a particular segment of memory. Different program modules can specify the same segment names in order to collect similar pieces of program into one named segment.

Several segment names have been predefined by Intel for use with their high level language compilers. These segment names are generated by the compiler according to the model (Small, Compact, Medium or Large) selected for the program compilation.

DATA	Data segment	(see class *'DATA'*)
STACK	Stack segment	(see combine type *STACK* and class *'STACK'*)
MEMORY	Memory segment	(see combine type *MEMORY* and class *'MEMORY'*)

Align Type

The segment align type determines the manner in which the segment is located in memory. The following align types have been predefined by Intel and adopted by others.

PARA Paragraph aligned (Default); least significant hex digit (4 bits) of the absolute address is *0H*.

BYTE Can be at any absolute address

WORD Even absolute address

PAGE Page aligned; two least significant hex digits (8 bits) of the absolute address are *00H*.

Combine Type

The segment combine type determines the manner in which the segment can be combined, if at all, with other segments by the linking loader. By default, a segment cannot be combined with others unless explicitly declared. The following combine types have been predefined by Intel and adopted by others.

`PUBLIC` Concatenate with other segments having the same segment name.

`COMMON` The segment and all others of the same segment name overlap and share a common base address. The largest of the named segments determines the final segment size.

`AT expression`
The segment is located at absolute address = `expression*16`.

`STACK` Concatenate with other segments of the same segment name in the segment reserved for use as a runtime stack.

`MEMORY` Place the segment in a special `COMMON` segment above all other segments.

In an AMX system, all stack segments must be specifically declared. Private AMX stacks and stacks for predefined tasks are allocated in your System Configuration Module. All other task stacks are allocated when tasks are created. No general "runtime stack" is required by AMX. However, many linking loaders insist that at least one of the linked program modules contain a segment with name `STACK`, combine type `STACK` and class `'STACK'`.

Therefore, you may have to create a dummy segment of length two with combine type `STACK` to avoid a linker error message. This will not be necessary if you are linking with the startup module of a high-level language compiler.

Class

The class of a segment is an arbitrary name, imbedded in single quotes, used to identify the purpose of a particular segment. The class name provides the linking loader with another way of collecting (classifying) similar segments besides by segment name. Several class names have been predefined by Intel and adopted by others. These include:

'CODE'	Program instruction sequences
'DATA'	Variables manipulated by a program excluding local variables maintained on a stack
'STACK'	Program stack
'CONST'	Program constants
'MEMORY'	Memory available for dynamic allocation and use by programs

KADAK has extended these predefined class names to include:

'AMXD'	Private AMX data
'AMXS'	Private AMX stacks and Large model task stacks

Group

A group is a collection of segments which are to be allocated to lie within a contiguous 64K byte region of memory. Each group must be defined with a unique group name. Intel has predefined one group name for use by its high level language compilers. The data group *DGROUP* is declared by the compiler to include the *DATA* segment, *STACK* segment and *MEMORY* segment when compiling programs under the Small or Medium compilation model.

Compilation Model

A compilation model is a set of conventions governing the use of the 8086 segment registers. Figure 1.6-1 illustrates the compilation models recommended by Intel for use with the 8086. AMX supports the Large model and the Medium model with extended pointers.

Figure 1.6-2 describes the segment naming conventions recommended for use with AMX 86. These naming conventions are compatible with Microsoft's MASM Assembler and C Compiler and several other C compilers.

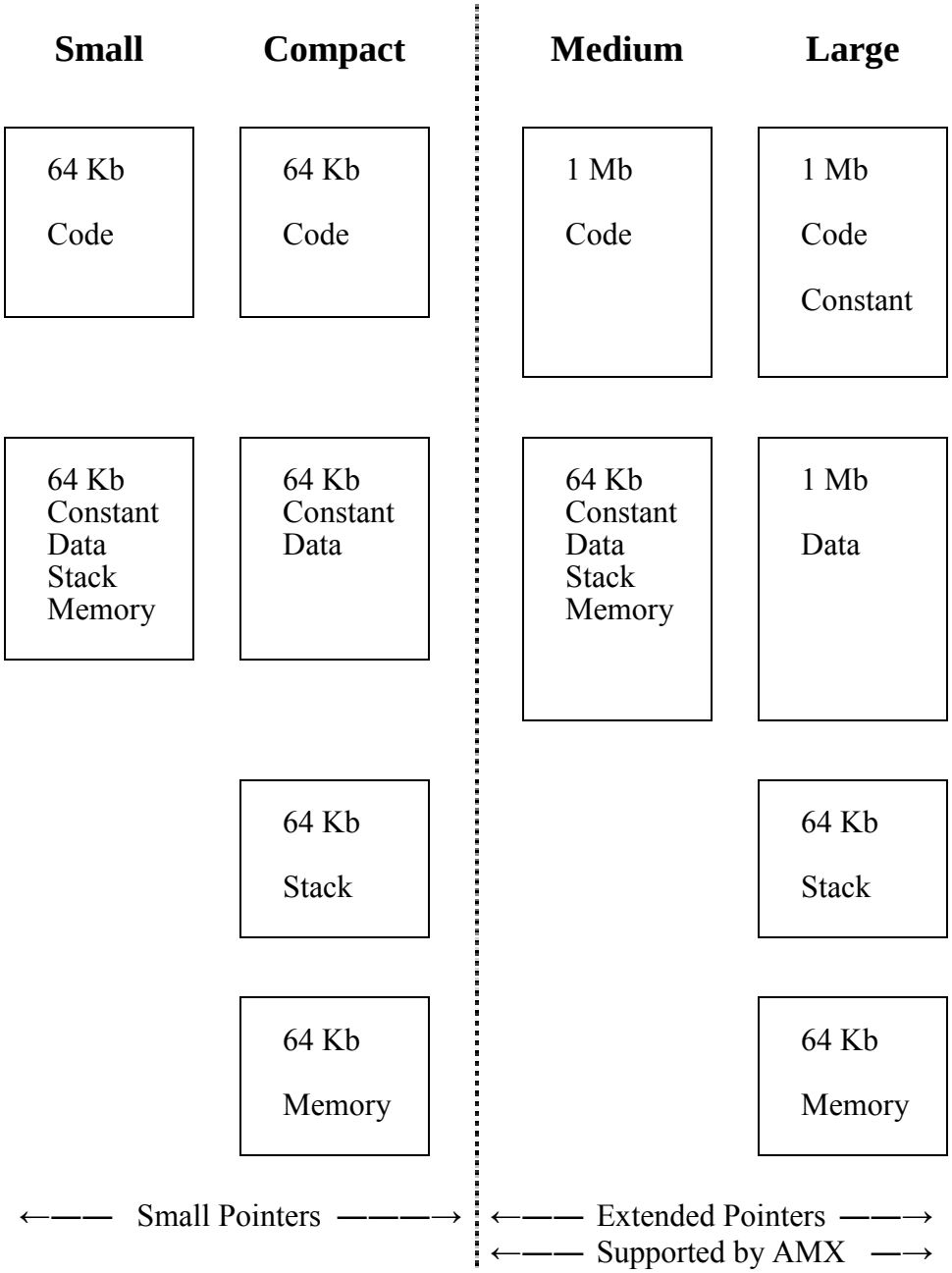

Figure 1.6-1 8086 Compilation Models

Segment Name	Description	Align Type	Combine Type	Class Name	Group Name
AMXCODE	All AMX code	*BYTE*	*PUBLIC*	*'CODE'*	none
AMXCODE	Configuration parameters	*PARA*	*PUBLIC*	*'CODE'*	none
AMXDATA	AMX private data	*PARA*	*PUBLIC*	*'AMXD'*	none
AMXMES	Message envelopes	*PARA*	none	*'AMXD'*	none
AMXSTAK	AMX private stacks	*PARA*	*PUBLIC*	*'AMXS'*	*none* (Note 1)
AMXDGRP	AMX *DGROUP* Data	*PARA*	*PUBLIC*	*'DATA'*	*DGROUP*
AMXVAR	Id Variables	*WORD*	*PUBLIC*	*'DATA'*	*DGROUP*

Note 1: Segment *AMXSTAK* will not exist if any Restart or Timer Procedures use the Medium model. The private AMX stacks will be located in segment *AMXDGRP*.

Figure 1.6-2 AMX Segment Naming Conventions

1.7 AMX Segment Prefix Module

AMX supports the Intel Medium and Large compilation models with *FAR* pointers. Many popular C compilers also support these models. However, since many constants and all global variables are often placed by the compiler in a data segment which is then declared a member of group *DGROUP*, the compiler effectively limits the data segment to 64Kb maximum.

The Microsoft Object Linker (LINK) organizes segments in memory according to the following criteria.

1. Segments with the same class identifier are collected together. The classes appear in memory in the order in which the classes are encountered by the linker.

2. Segments with the same class identifier appear sequentially in memory according to the order in which they were encountered by the linker.

3. Segments with the same segment name and class identifier which are declared *PUBLIC* appear contiguously in memory in the order in which they are encountered by the linker. Note that segments combined with the *PUBLIC* attribute will reside in one canonical frame (Microsoft's term); offsets in these segments will be relative to a shared segment base address.

Segment class, name and combine attribute are the only criteria used by the linker to establish the order of segments in memory. Once the order has been established, each segment is allocated to a canonical frame which determines the base address for the segment. All references within the segment are with offsets relative to the assigned base address. Remember that all *PUBLIC* segments of the same name and class share a common base address.

The Microsoft MASM Assembler and most C compilers also support the group concept. Segments can be declared to be members of a group. Groups DO NOT affect the order in which segments are loaded in memory. Groups simply provide another rule used by the linker in establishing the canonical frames by which segments are referenced.

If several segments are declared to be members of a group, say *DGROUP*, then the base address for all of these segments will be the same. The base address is determined by the location of the group member which is lowest in memory.

Serious linking problems can occur because of grouping effects. Remember that membership in a group does not affect the order in which segments are linked. That is determined by segment class, name and combine attribute. Therefore you can produce a segment order by virtue of poor class and segment name choices which force segments belonging to a group to be more than 64K apart in memory. The segments at the top of the group cannot be referenced relative to the base address of the group member which is lowest in memory. The linker therefore produces its "fixup" error message when it tries to resolve such a reference.

The startup module for most C compilers assumes that the stack segment will be the last segment loaded by the linker and hence will reside at the top of your load module memory image. Then, using this assumption, it allocates all memory above the stack segment to the C heap for management by the C library memory allocator.

This assumption may present difficulties in establishing your code, data, stack and memory partitions in your AMX system. The AMX Segment Prefix Module *AJ831PRF.ASM* overcomes these problems if it is linked as the first module in your system. It defines several dummy segments with standard AMX class identifiers to force the following segment ordering in the AMX system.

Segment Name	Class	Purpose
AMXCODE	*'CODE'*	Collect all code segments for most C compilers.
AMXCODT	*'TEXT'*	Collect other C compiler's code segments.
AMXDATA	*'AMXD'*	AMX private data
AMXSTAK	*'AMXS'*	AMX Large model stacks

The order of segments in this list is the preferred order of segments from low memory to high memory. Segments of class *'CODE'* and *'TEXT'* can be in ROM. All other segments must be in RAM.

Note that the segment names are also alphabetically ordered. This is not accidental. Some assemblers order the segments in a module alphabetically in the object file. This is not necessarily the order in which they appear in the source file. It is the order of segments in the object module that determines the order in which they are encountered by the linker. Therefore, all bases are covered by ordering the segments in the source file both alphabetically and logically.

The order of segments following these predefined AMX segments will vary according to the dictates of your C startup module.

There should be no segments following the stack segment of class *'STACK'* provided by the C startup module or else they will be treated as heap by most C startup modules.

2. AMX Programming Hints

2.1 C Code Portability

If you are coding in C and expect 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 to be readily ported to 16-bit processors.

Do not use unions to extract *char* or *short int* values from *long* or pointer variables. The byte reversal of 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 Chapter 4 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.

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

2.2 AMX Stack Allocation

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

1. Start with the minimum task stack size required by AMX. This stack is used to store the task state when it is interrupted. The minimum stack allows a task procedure with no local variables to call any AMX procedure and return to AMX.

2. Determine the stack size for each procedure called by the task. Most C procedures will require a minimum of eight bytes for storage of return address and saved registers plus storage for all calling parameters and automatic variables.

3. Find the procedure nesting path that requires the most stack space and add the stack sizes of each procedure to the minimum task stack. Recursive procedures must have a stack size large enough to allow for the maximum recursion depth.

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

Stack Checking

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

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

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

2.3 Choosing a Synchronization Method

Task Wait/Wake

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

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

The task id can be used as a convenient boolean indicator. Create an event variable of type *AMXID* and set it to *0* to indicate that no task is waiting for the particular event. When a task is about to wait for the event, it can set the event variable to the task's id (using *ajtkid*) thereby informing the event handler that a task is waiting and identifying the task at the same time.

Task Trigger

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

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

Counting Semaphore

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

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

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

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

Resource Semaphore

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

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

Mailbox or Message Exchange

The AMX task mailbox and message exchange offer information passing synchronization methods. The event handler creates a message which it sends to a task mailbox or message exchange. A task automatically receives messages posted to any of its mailboxes. Any task can ask for or wait for a message from a message exchange, thereby synchronizing with the event handler which generated the message.

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

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

Message Tasks

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

Ack-Back Messages

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

Event Group Flags

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

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

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

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

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

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

2.4 AMX Caveats

Task Trigger vs Message Queuing

Do not trigger AMX message tasks. Tasks with private message mailboxes are automatically triggered by AMX.

Message Envelopes

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

AMX Message Length

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

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

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

AMX Shutdown

If you use the AMX procedure *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.

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

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

Code Size and Speed

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

If possible, use task mailboxes instead of message exchanges. Message exchanges require more memory and are slightly slower to use.

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

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

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

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

Do not use `ajtmcnv` in ISPs or Timer Procedures to convert milliseconds to AMX ticks. This procedure is relatively slow. Compute the value at launch time or in some task and make the value permanently available in a private variable.

Accessing the AMX User Parameter Table

Many C compilers place constants, global variables and initialized global data in one or more segments which are declared to be of combine type `PUBLIC` and class `'DATA'`. This segment is declared to be a member of group `DGROUP`.

This approach to global variables precludes direct access by tasks to any global variable such as your User Parameter Table in your System Configuration Module. This variable cannot be directly accessed from C because it resides in the code segment which is not a member of group `DGROUP`.

This particular problem is resolved by using AMX procedure `ajupt` to fetch a pointer to the User Parameter Table.

This page left blank intentionally.

3. Debugging an AMX Application

3.1 Breakpoints and Tracing

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

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

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

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

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

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

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

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

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

- AMX took a fatal exit and unconditionally halted (see the next topic regarding Fatal Exit Procedures).
- Your AMX Configuration Module contains invalid or unresolved information which leads to improper AMX operation. (This will be unlikely if you used the Configuration Manager and Generator to create your module.)
- You failed to include an AMX option in your configuration which is vital to AMX success. For example, you expect to use AMX timing features but you have not included a clock ISP of any kind.
- 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.
- Your Restart Procedures caused the AMX Kernel Stack to overflow. You must not sprinkle `printf` statements in Restart Procedures for testing purposes.
- 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 test program code "borrowed" and adapted for a new application.

Using a Fatal Exit Procedure

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

If you have not 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.

3.3 Debugging Caveats

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

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

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

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

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

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

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

4. Sample AMX System

A sample AMX system is provided with AMX to illustrate the ease with which an AMX system can be created. The following files are provided for each toolset supported by KADAK. Other target sensitive files including a clock driver and serial driver are also provided.

`AM831SCF.UP`	User Parameter File
`AM831SAM.C`	Sample System
`AM831SAM.LKS`	Link Specification File

Operation

The system includes two Restart Procedures, a Print Task, a Shutdown Task, a clock ISP, two timers and an Exit Procedure.

The Print Task and both timers are predefined in the User Parameter File. The first Restart Procedure starts the two timers and sends a sign-on message to the Print Task for display. The clock ISP is installed and the clock is enabled by the clock driver's `chclockinit()` procedure which is included in the list of AMX Restart Procedures.

The two timers are periodic. Each sends a message to the Print Task for display. The message includes the value of the AMX tick counter at that instant.

The second Restart Procedure creates the Shutdown Task and triggers it. The task waits for one minute and initiates a shutdown of the AMX system.

All Exit Procedures execute in the context of the Shutdown Task. The sample Exit Procedure stops the two timers and sends a sign-off message to the Print Task for display. It then waits for acknowledgement that the message has been displayed.

The clock is disabled and the clock ISP is removed by the clock driver's `chclockexit()` procedure which is included in the list of AMX Exit Procedures.

Device I/O

AMX is delivered with clock drivers for several counter/timer chips often used 80x86 target processors. One of these clock drivers is used by the sample system. The Borland, Microsoft and WATCOM samples use the PC/AT Clock Driver. The 20-bit Paradigm sample uses the Intel 8253 (8254) Clock Driver. The 24-bit Paradigm sample uses the Am186ES Clock Driver.

You may have to rebuild the clock driver to match your hardware interface or provide an alternate clock ISP as described in Appendix C of the AMX 86 Tool Guide.

A console output device is required. For purposes of illustration, an artificial serial I/O interface will be invented. The device will be used in a polled mode without interrupts. It is assumed that the device is initialized correctly (start bits, stop bits, baud, etc.) by writing the bit pattern `0x3E` to port address `0x0F2D`. The device is considered ready for output if bit 7 of input port `0x0F2D` reads non-zero. Characters are written to device address `0x0F2F`.

AMX is delivered with serial drivers for several UART chips often used with the 80x86 target processors. One of these serial drivers is used by the sample system. The Borland, Microsoft and WATCOM samples use the PC/AT Console Driver. The 20-bit Paradigm sample uses the Intel 8250 Serial Driver. The 24-bit Paradigm sample uses the Am186ES Serial Driver.

Construction

The AMX Sample System is ready to be constructed using any of the toolsets supported by KADAK.

Use the AMX Configuration Builder to view the User Parameter File *AM831SCF.UP*. From the File menu, select Generate... to create the System Configuration Module *AM831SCF.ASM*.

Review the AMX Tool Guide for instructions for compiling, assembling, linking and locating modules. Refer to the relevant chapter for the tools which you are using.

Assemble the System Configuration Module file *AM831SCF.ASM* according to the instructions provided in the AMX Tool Guide.

Compile the Sample System file *AM831SAM.C* as a normal AMX application module as indicated in the Tool Guide. You may also have to edit the clock driver and serial driver modules to adapt them to your particular target hardware. You must then compile the clock driver and serial driver following the instructions provided in the AMX Tool Guide for compiling application C files.

Link and locate the Sample System using the link/locate specification files provided with AMX. Link and locate the sample program, the configuration file and all driver modules with the AMX Library and your C run-time library. Follow the instructions in the AMX Tool Guide for linking a system with your toolset.

If you are using Borland, Microsoft or WATCOM tools, you can run the Sample System program *AM831SAM.EXE* from the DOS command prompt of a standard DOS system (not the Windows® MS-DOS command prompt).

If you are using Paradigm tools, transfer the load module *AM831SAM.AXE* (or equivalent) to your target hardware using the Paradigm debugger accessible via the Paradigm IDE.

> Note
>
> See batch file *AM831SAM.BAT* in toolset directory *TOOLxx* for a complete example of this construction process.

```
/* Restart Procedure 1                                                  */

void cdecl rr1(void)
{
      concfg();                         /* Configure I/O device       */

      ajtmwr(tmr2id, 1L);
      ajtmwr(tmr1id, 1L);               /* Start timers               */

/* Note: The hardware clock is initialized and started by             */
/*    procedure chclockinit() in the AMX clock driver.                */
/*    Procedure chclockinit() is included in the list of              */
/*    Restart Procedures in the User Parameter File AM831SCF.UP.      */

      ajsendp(prtskid, 0, "AMX Sample System begins.\n\n");
      }


/* Timer Procedure 1                                                  */
void cdecl timer1(void)
{
      ajsendp(prtskid, 3,
      "Timer 1 at %lu ticks (mailbox 3)\n", ajtick());
      }


/* Timer Procedure 2                                                  */
void cdecl timer2(void)
{
      ajsendp(prtskid, 2,
      "Timer 2 at %lu ticks (mailbox 2)\n", ajtick());
      }


/* Print Task                                                         */

void cdecl prtask(int msg)
{
      char  buffer[80];
      char  *cp;
      struct prmsg {
            char  *fmtsp;               /* Format string pointer      */
            long  param;                /* Parameter                  */
            } *msgp;

      msgp = (struct prmsg *)(&msg);    /* Get pointer to message     */

                                        /* Format string into buffer  */
      sprintf(buffer, msgp->fmtsp, msgp->param);

                                        /* Print the string           */
      for (cp = buffer; *cp != '\0'; cp++) {
            conout(*cp);
            if (*cp == '\n')
                  conout('\r');
            }
      }
```

```
/* Simple serial I/O device                                          */

#define CONCFGR (0x0F2D)              /* Configuration register      */
#define CONSTAT (0x0F2D)              /* Status register             */
#define CONDATA (0x0F2F)              /* Data register               */


/* Configure serial I/O device                                       */

void concfg(void)
{
      ajout8(CONCFGR, 0x3E);          /* Configure serial port       */
      }


/* Output character                                                  */

void conout(char ch)
{
      while ( (ajin8(CONSTAT) & 0x80) == 0)
            ;                         /* Wait for ready              */

      ajout8(CONDATA, ch);            /* Write character             */
      }
```