AMX/FS™ 86

File System

USER'S GUIDE

First Printing: July 1, 1995
Last Printing: November 1, 2007

TECHNICAL SUPPORT

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

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

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

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

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

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

DISCLAIMER

TRADEMARKS

AMX/FS 86 USER'S GUIDE
Table of Contents

Table of Figures

This page left blank intentionally.

1. AMX/FS Overview

1.1 Introduction

The AMX/FS™ File System is a full-featured, high performance MS-DOS® compatible file system for use with KADAK's AMX™ Multitasking Executive in embedded applications. AMX/FS is provided as a library of C procedures which permit multiple tasks to concurrently manipulate files.

A RAM Disk device driver is provided to permit processor memory to be used for data storage and retrieval using file operations.

The User Device Driver (UDD) provides a template for customizing your own device driver. The UDD supports both single partition removable media controllers and multiple partition fixed media controllers.

For AMX 86 applications running on a conventional PC, A PC BIOS device driver is provided to permit the AMX/FS File System to use the PC BIOS floppy and hard disk device I/O services. Since the BIOS code executes only in real-mode on PC platforms, the AMX/FS PC BIOS device driver is only offered with AMX 86.

Optional floppy and IDE disk device drivers, ready for use with PC compatible hardware interfaces, are available as separate products. These drivers can be readily ported to meet your specific device requirements. For convenience, these drivers are described in this manual even though the products are available separately.

Installation of the AMX/FS File System is a simple process. AMX/FS is provided in library form ready for use with any of the software development toolsets supported by KADAK. No AMX/FS library construction or porting is required. However, make files are included to permit construction of the product should you wish to do so.

The AMX/FS File System is easily configured to meet your particular disk requirements using the AMX Configuration Generator. Simple commands in your AMX User Parameter File instruct the Generator to create an AMX/FS Configuration Module which defines your logical disk drives and selects the RAM Disk, UDD, PC BIOS, floppy or IDE device drivers in various combinations. Configuration errors are detected and reported before you can even construct your application.

The AMX/FS File System is based on the RTFS FAT File System Software created by Etc Bin Systems. This file system, first released in 1989, is in use in embedded applications world wide. The file system has been ported for use with AMX and has undergone extensive testing to ensure that it meets KADAK's exacting standards for reliability and maintainability. What sets the AMX/FS File System apart from all other implementations is the simplicity with which it can be incorporated into your application.

The purpose of this manual is to provide the system designer and applications programmer with the information required to properly configure and implement an AMX-based real-time application using the AMX/FS File System.

It is assumed that you have a basic knowledge of file system fundamentals. It is also assumed that you are familiar with the architecture of the processor on which you will be using AMX. It is further assumed that you are familiar with the rudiments of microprocessor programming including the concepts of code, data and stack separation.

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

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

Manual Summary

Chapter 1 of this manual describes the AMX/FS File System and how it is used.

Chapter 2 describes the AMX System Configuration Builder and the manner in which it is used to create your AMX/FS Configuration Module.

Chapter 3 is the application programming guide. It provides detailed descriptions of the AMX/FS service procedures which are available in the AMX/FS Library or with specific device drivers.

Chapter 4 provides details concerning device drivers and their use.

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

AMX/FS Tool Guide

This manual describes the use of AMX/FS for all target processors. Target specific requirements or programming considerations are provided in separate appendices.

This manual describes the use of AMX/FS in a tool set independent fashion. References to specific assemblers, compilers, librarians, linkers, locators and debuggers are purposely omitted.

The **AMX Tool Guide** provides guidance for the proper use of AMX with each toolset with which AMX has been tested. The AMX Tool Guide is located at the front of your AMX Reference Manual. The guide instructs you in the use of the tools to build your AMX application. The instructions apply equally to your AMX/FS application.

A separate **AMX/FS Tool Guide** describes the few toolset dependent requirements which are unique to the AMX/FS software development process. It also describes the procedure to be followed to reconstruct the AMX/FS Library should you wish to do so.

1.2 Glossary

AMX Glossary   Refer to the AMX Glossary in Chapter 1.2 of the AMX User's
                      Guide for a list of all AMX related terms.

Blocking Buffer   One of a set of private data buffers used by AMX/FS to control
                      access to a disk drive during directory searches or manipulation.

Counting Semaphore

                      A particular type of AMX semaphore used by an AMX/FS device
                      driver Interrupt Service Procedure to signal completion of a device
                      operation.

Current Working Directory

                      The directory on a logical drive which, in the absence of any
                      specific directory specification, AMX/FS will use to resolve an
                      otherwise ambiguous file or directory reference.   AMX/FS
                      maintains a separate current working directory for each logical
                      drive for each registered task.

Default Drive   The logical drive which, in the absence of any specific drive
                      specification, AMX/FS will use to resolve an otherwise ambiguous
                      file or directory reference. AMX/FS maintains a separate default
                      drive for each registered task.

Drive Id   The upper case alphabetic letter (`A` to `Z`) used to identify one of the
                      26 possible logical drives supported by AMX/FS. Lower case
                      letters are not valid drive ids.

Drivename   The name of a logical drive consisting of the single character drive
                      id followed by a colon as in `C:`.

Drive Number   The positive integer (0 to 25) used to identify one of the 26
                      possible logical drives supported by AMX/FS.

Error Code   A series of signed integers used by AMX/FS to indicate error or
                      warning conditions detected by AMX/FS service procedures.

Exit Procedure   An AMX or application procedure executed by AMX during the
                      exit phase when an AMX system is shut down.

Fatal Trap   A trap taken by AMX/FS when it has detected a condition which is
                      considered so abnormal that to proceed might risk catastrophic
                      consequences. AMX/FS reports the error and loops at its breakout
                      procedure `fjfsfatal()`.

Fault Trap

A trap taken by AMX/FS when it has detected a condition which, although unusual and possibly abnormal, still permits AMX/FS to function. AMX/FS reports the error and calls its breakout procedure *fjfsfault()*.

Filebase

The primary one to eight character name given to an MS-DOS compatible file.

File Extension

The secondary zero to three character name given to an MS-DOS compatible file. The file extension, if one exists, is separated from the filebase name by a period.

File Handle

An identifier assigned by AMX/FS for use by your application to reference an open file.

Filename

An MS-DOS file name consisting of a filebase and an optional file extension.

Filepath

A text string giving a description of the location of a directory on a logical drive. The description can be ambiguous in which case the default drive and/or the drive's current working directory must be used to resolve the reference.

FS Configuration Module

A software module, produced by the AMX Configuration Generator utility, which defines the AMX/FS parameters, device drivers and logical drives of a particular AMX/FS application.

Full Filename

A text string consisting of a drivename, a rootpath and a filename giving the complete, unambiguous location and name of a file on a logical drive.

Fullpath

A text string consisting of a drivename and a rootpath giving the complete, unambiguous location of a directory on a logical drive.

Logical Drive

A drive partition which is accessible by AMX/FS. Logical drives are defined in the AMX/FS Configuration Module.

Physical Drive

A single disk unit attached to or embedded in a disk controller. The storage on a physical drive may be subdivided into one or more regions called partitions.

Partition

A region of the storage area on a physical drive. A partition in MS-DOS format can be accessed by AMX/FS as a logical drive.

RAM

Alterable memory used for data storage and stacks.

RAM Disk | A region of memory treated like a partition on a physical drive in MS-DOS format and hence accessible by AMX/FS as a logical drive.

Registered Task | A task which has declared its intent to make use of AMX/FS. A task must register itself before it can make use of AMX/FS services.

Renounce | Declare no further intention to use AMX/FS. A task can renounce its use of AMX/FS if the task has no further need to use any AMX/FS services.

Resource Semaphore

A particular type of AMX semaphore used by AMX/FS to provide mutually exclusive access to files, physical disk drives, device drivers and critical sections of code within AMX/FS.

ROM | Read only memory of all types including PROMs and EPROMs.

Rootpath | A text string giving the complete, unambiguous list of directories from the root directory of a logical drive to a particular directory on the drive.

Segment | An area of memory in which AMX/FS code or data is stored. Segments are sometimes called sections or regions according to the nomenclature adopted for a particular processor.

1.3 AMX/FS Nomenclature

The following nomenclature standards have been adopted throughout the AMX/FS reference manual.

Numbers used in this manual are decimal unless otherwise indicated. Hexadecimal numbers are indicated in the format $0xABCD$. The terminology $A(Table\ XYZ)$ is used to define addresses. It is read as "the address of Table XYZ". Read/write memory is referred to as RAM. Read only memory (non-volatile storage) is referred to as ROM.

AMX/FS symbol names and reserved words are identified as follows:

`fjkkpppp`	AMX/FS C procedure name `pppp` for service of class `kk`
`fjxtttt`	AMX/FS structure name of type `tttt`
`xttttyyy`	Member `yyy` of an AMX/FS structure of type `tttt`
`FJ_EXXXXX`	AMX/FS Error Code `XXXXX`
`FJ_ssssss`	Reserved symbols defined in AMX/FS header files
`fj_ssssss`	Reserved AMX/FS symbols
`PC_ssssss`	Reserved AMX/FS symbols
`pc_ssssss`	Reserved AMX/FS symbols
`FJnnnFFF.XXX`	AMX/FS filenames
`FJZZZ.H`	Generic AMX/FS include file

In addition to those symbols listed above, the following AMX 86 and 32-bit AMX symbol names and reserved words are also referenced:

`AMX_ID`	AMX object identifier
`AJ_ssssss`	Reserved AMX symbols
`ajssssss`	Reserved AMX symbols
`CJ_CCPP`	Procedure uses C parameter passing conventions
`CJ_CCHUGE`	A normalized 80x86 huge pointer
`CJ_NULL`	A `NULL` data pointer
`CJ_NULLFN`	A `NULL` function pointer

Since this guide describes the use of AMX/FS on any target processor, the explicit 3-digit KADAK part numbers cannot be used. Therefore, AMX and AMX/FS part numbers are replaced by the strings `mmm` and `nnn` respectively. For example, the AMX 86 installation subdirectory `AMX831` may be referred to as `AMXmmm` and the AMX/FS 86 Library module `FJ838.LIB` may be referred to as file `FJnnn.LIB`.

When AMX and AMX/FS are installed, generic header files `FJZZZ.H` and `FJZZZINC.H` are created from their part numbered counterparts. Hence, when you install AMX/FS 86, the generic file `FJZZZ.H` will be a copy of file `FJ838.H`. By referencing the generic header file in your application C source file, the appropriate target-dependent, part numbered AMX and AMX/FS header files are automatically included and your application becomes readily portable to other target processors.

1.4 Installation and Use

Installation

The AMX/FS File System is delivered in source and library form ready for use with the software development toolsets which KADAK supports.

Installation instructions are provided in file `README.TXT` on the product disk. You will be given the opportunity to install AMX/FS on the drive of your choice, ready for use with any of the toolsets supported by KADAK. You can install the AMX/FS File System for use with all supported toolsets if you so desire.

If you have purchased any of the AMX/FS options such as the AMX/FS Floppy Driver or IDE Driver, the options will also be installed into the AMX/FS installation directory. The options are provided in source form ready for use with each of the supported toolsets.

General Features

The AMX/FS File System can be used with any disk drives which are recorded in MS-DOS format. AMX/FS supports drives like floppy disks which use a removable media. AMX/FS also supports hard disks with a fixed media which can be partitioned into one or more logical drives. The AMX/FS File System also includes a RAM Disk driver which permits a region of memory to be managed as though it were a disk.

A logical drive, also called a volume, must be opened (mounted) before any task can access it. Once opened, the drive becomes accessible to all tasks until it is closed (unmounted).

Once a logical drive is open, the directories and files on it become accessible. AMX/FS provides procedures to open and close drives and measure their usage (free space). Each task can set its own default drive. In case of serious trouble, AMX/FS can be directed to abort all operations on a particular drive.

AMX/FS offers a full set of directory access services. Each task can establish its own current working directory for each available logical drive. AMX/FS will use a task's default drive and current working directories to resolve any ambiguous file or directory references which the task makes.

Directories can be created and deleted (removed). A filepath can be tested to determine if it actually references a directory.

AMX/FS offers a wide range of file manipulation services. Files can be created, opened, read, written, closed and deleted (unlinked). A file, once open, is referenced by a task using a file handle provided to the task by AMX/FS at the time the file was opened. A file is created by opening it with an access mode that permits AMX/FS to create the file if it does not already exist. A variation of the open request permits you to test if a file exists without creating a new file if it does not exist. Services also exist to permit a search for all of the files in a directory which match a particular file name pattern of interest.

File status can be acquired for an open file or for any file simply referenced by name. The status includes the file's name (filebase and extension), attributes and most recent modification time and date. A file's attributes can also be accessed and, with care, modified if so desired.

The read and write position within a file is determined by the file pointer maintained by AMX/FS for every open file. The position of the file pointer can be tested and manipulated. A file can also be truncated (shrunk) or extended (grown).

Finally, AMX/FS provides a limited set of string formatting procedures which can be used, if needed, to replace C library functions which may be non-reentrant or otherwise unsuitable for embedded system development. Simple replacements for *sprintf*, *itoa* and *ltoa* are provided.

Disk Formatting

In order for AMX/FS to access a disk drive, the disk media must be formatted. Disk formatting is a two or three step process. First, the media must have low level sector access information recorded on every track (cylinder). For removable media like floppy diskettes, this low level format can be done by the floppy drive controller. For fixed disks, the low level information is recorded by the drive manufacturer.

Once sectors can be accessed, the drive can be partitioned into one or more logical drives. Removable media like floppy diskettes are not partitioned. The diskette contains no partition information. Fixed disks are partitioned and, even if there is only one partition, the disk will contain partition information.

AMX/FS does not provide services to allow you to partition a fixed drive. The drive must be partitioned before it can be accessed by AMX/FS. It is recommended that a drive which is to be used with AMX/FS be attached to a conventional PC and partitioned using an MS-DOS drive partitioning software utility. The drive can then be attached to the disk controller in the target hardware environment.

Once low level formatted and, if necessary, partitioned, the logical drive must have a valid MS-DOS file system recorded on the drive. This process, called making an MS-DOS file system, is supported by AMX/FS procedure *fjmkfs()*.

The AMX/FS Floppy Driver includes a procedure *fjfmtfloppy()* which performs two levels of diskette formatting. First, it uses the floppy disk controller to record the low level sector information onto the diskette. It then calls *fjmkfs()* to make an MS-DOS file system on the diskette.

The AMX/FS File System includes a RAM Disk driver which permits a region of memory to be used as a disk storage device. The RAM Disk has no low level format. The RAM Disk driver includes procedure *fjfmtram()* (and *fjfmtxram()*) which initializes the RAM Disk for use by calling *fjmkfs()* to make an MS-DOS file system in the region of memory reserved for its use.

This page left blank intentionally.

1.5 Task Registration

The AMX/FS File System permits multiple tasks operating under AMX to concurrently access files. Every task which wishes to use AMX/FS services must first register as an AMX/FS user. When a task no longer requires access to AMX/FS services, it can renounce its use of AMX/FS. AMX/FS service procedures *fjfssignin()* and *fjfssignout()* can be called by tasks for this purpose.

For every registered task, AMX/FS maintains a private User Access Block (UAB). The maximum number of concurrently registered user tasks is a configuration parameter which you can adjust to meet your application requirements (see Chapter 2.3). When a task is registered, a UAB is allocated to the task. When a task renounces its use of AMX/FS, its UAB is released and made available for use by other tasks.

A task remains registered until it renounces its use of AMX/FS. A task, once registered, can request registration again but will simply continue to use the UAB already allocated to the task. Hence, a task can register as a user every time it begins execution in response to a task trigger or receipt of an AMX message.

For convenience, AMX/FS will automatically register a task, if it is not already registered, whenever the task requests an AMX/FS service for which registration is a prerequisite. This feature makes using AMX/FS easy. If you know by design that only four of your ten tasks will ever require file access, configure your AMX/FS file system to allow only four concurrent user tasks. The four tasks can then use AMX/FS without registration knowing that AMX/FS will automatically register them successfully.

Default Drive

AMX/FS maintains a default drive for each registered task. The default drive is set to logical drive number *0*, drive *A*, whenever a task is first registered. A task can change its default drive at will. AMX/FS will always use the current task's default drive when it must resolve an ambiguous file reference.

Current Working Directory

For each registered task, AMX/FS maintains a list of current working directories for all available logical drives. Each drive's current working directory is set to the drive's root directory (\.) whenever a task is first registered. A task can change its current working directory for any drive at will. AMX/FS will always use the current task's current working directory for a particular drive when it must resolve an ambiguous path in a file reference.

1.6 Task Error Handling

DOS and UNIX file systems provide a public variable *errno* which always contains the error code generated by the most recent file operation requested by the application. AMX/FS maintains the equivalent error code for each registered task. The AMX/FS error number is NOT a public variable. It can only be accessed by a task through a call to AMX/FS procedure *fjfserrno()*. A task can only access its own error number. Procedure *fjfserrno()* always returns *0* if no error has been detected. Otherwise, it returns an error code *FJ_Exxxx* describing the reason that the task's most recent call to AMX/FS failed.

In addition to the application level error number maintained for each task, AMX/FS also records zero, one or two lower level error codes which may be of use in isolating the exact cause of a particular file access fault.

This detailed AMX/FS error information can only be accessed if you have configured your AMX/FS file system to enable error reporting, sometimes called logging. By default, error reporting is always disabled and will only be available if you have explicitly enabled error reporting via directive *...FSYS* in your AMX/FS Configuration Module (see Chapter 2.3).

Task Error Handler

The detailed AMX/FS error information is accessible in string form by a task's Error Handler. When a task is first registered, it has no Error Handler. Once registered, a task can install an Error Handler with a call to AMX/FS procedure *fjfserrfn()*. Once installed, the task's Error Handler remains in effect until the task replaces it with a different Error Handler or cancels the handler by installing the null Error Handler *CJ_NULLFN*.

The task's Error Handler is invoked by the task itself with a call to AMX/FS procedure *fjfsperror()* to report (log) the most recently recorded error information.

The task Error Handler is an application procedure which is prototyped as follows.

```
void CJ_CCPP errhandler(int nmsg, char *m1, char *m2,
                                  char *m3, char *m4);
```

The Error Handler receives an integer *nmsg* which indicates how many of the string pointers *m1* to *m4* contain valid strings. If *nmsg* has value *1*, only pointer *m1* is valid. If *nmsg* is *2*, both *m1* and *m2* reference valid strings.

When a task calls *fjfsperror()*, it can specify an optional application error message string. If such a string is provided, it will always be received by the task's Error Handler as parameter *m1*.

The next valid string received by the Error Handler will always describe the application error reported by *fjfserrno()* as error code *FJ_Exxxx*. Following that will be a string, if one is available, describing the AMX/FS file system fault or a driver level fault. Following that will be a string, if one is available, describing a device level fault encountered by a driver.

All strings presented to the Error Handler are null (`\0`) terminated. With the possible exception of the task's application string, all of the strings have only printable characters. The strings provided by AMX/FS are located in AMX/FS file *FJ838ERR.C*.

The following example illustrates a task Error Handler which records the strings in an AMX circular list. Note that this approach is only valid if all application strings presented to *fjfsperror()* by the task are static or external and unalterable.

```
#include "FJZZZ.H"

#define NERRORS 32

struct errlist {                            /* Error list              */
    struct cjxclist erlhead;
    char     *erlslots[NERRORS];
    } errorlist;

extern AMXID mytaskid;

void CJ_CCPP myrestart(void)            /* Restart Procedure       */
{
                                        /* Initialize error list   */
    ajrstl(&errorlist, sizeof(errorlist.erlslots[0]), NERRORS);

    ajtrig(mytaskid);
    }


void CJ_CCPP myhandler(int nmsg,        /* Task Error Handler      */
         char *m1, char *m2, char *m3, char *m4)
{
    if (nmsg >= 1)
            ajabl(&errorlist, m1);
    if (nmsg >= 2)
            ajabl(&errorlist, m2);
    if (nmsg >= 3)
            ajabl(&errorlist, m3);
    if (nmsg >= 4)
            ajabl(&errorlist, m4);
}


void CJ_CCPP mytask(void)
{
    fjfssignin();                           /* Register as a user task */
                                            /* Install Error Handler   */
    fjfserrfn((FJ_CALLBACK)myhandler);

    if (fjdrvopen("A:") < 0)
            fjfsperror("Cannot mount drive A:");

    else if (fjdrvclose("A:") < 0)
            fjfsperror("Cannot unmount drive A:");
    }
```

This page left blank intentionally.

1.7 File Name Conventions

The AMX/FS File System uses the MS-DOS file naming conventions. Although most users will be familiar with DOS file names, they are described here in order to introduce terminology which can be used to reference the components of a file name unambiguously throughout the remainder of this manual.

The term **file name** is used as a rather loose term for any valid name which can be used to reference a file. The term file name does not give any indication of the form of the actual name of the file.

An MS-DOS **filename** consists of two parts: an 8-character name which we will call a **filebase** and an optional 3-character file **extension**. The filebase and extension are separated by a period. Valid filenames are:

```
Filebase
Filebase.
Filebase.ext
```

A file resides in a directory or a subdirectory on a logical drive. Directories are pseudo files which act as placeholders for files and other directories called subdirectories. A directory name, which we will call a **dirname**, looks exactly like a filename.

Two special directories exist within every directory. The current directory (named .) and the parent (previous) directory (named ..).

The location of the file in a particular directory is determined by its path which, again, is a rather loose term. The unambiguous location of the file is determined by its **fullpath** which includes a drivename and rootpath.

The **drivename** is a 2-character name for a logical drive. The drivename consists of the drive id (an upper case letter *A* to *Z*) followed by the character *:*.

The **rootpath** defines the list of directories from the root directory on the logical drive through to the directory or subdirectory which contains the file. The rootpath begins with the file separator \ which is followed by zero or more dirnames separated from each other by the file separator \. The rootpath may or may not include a final file separator \.

Valid fullpaths are:

```
C:\
C:\.
D:\dirname
D:\dirname.\
E:\dirname\dir2name
E:\dirname\dir2name.ext
E:\dirname\dir2name.ext\
```

Armed with these definitions, we can now unambiguously locate a file using a **full filename**. The full filename includes a drivename, a rootpath and a filename.

Valid full filenames are:

```
C:\dirname\Filebase
D:\Filebase.
E:\Filebase.ext
```

Note that `C:dirname\Filebase` or `D:Filebase` are not full filenames because they do not include a valid rootpath. The rootpath's leading `\` separator is missing.

Path descriptions which do not unambiguously determine the location of a file are called **filepaths**. A filepath consists of an optional drivename and a partial list of zero or more dirnames separated by the `\` character.

AMX/FS uses the concept of a **default drive** to resolve a missing drive reference in an ambiguous filepath. If a filepath does not include a drivename, AMX/FS assumes that the file resides on the default drive in use by the current task.

AMX/FS uses the concept of a **current working directory** to resolve the directory referenced by an ambiguous filepath. If a filepath does not include a valid rootpath, AMX/FS appends the path information from the ambiguous filepath to the rootpath for the current task's current working directory on the logical drive of interest. In this manner, AMX/FS derives an unambiguous full filename.

The following examples illustrate this process.

Default drive is `E:`
Current working directory on drive `E:` is `\dirname`

Filepath	Resolves to
`E:Filebase`	`E:\dirname\Filebase`
`E:dir2name\Filebase`	`E:\dirname\dir2name\Filebase`
`..\Filebase`	`E:\Filebase`

Note

AMX/FS directory and file names are case sensitive.

---

2. AMX/FS Configuration

2.1 AMX/FS Configuration Module

The AMX/FS Configuration Module (also referred to as the FS Configuration Module) defines the manner in which AMX/FS will be used in your AMX system. The AMX Configuration Generator will create this module for you. This is the same tool used to create your AMX System Configuration Module (see Chapter 15.2 of the AMX User's Guide).

The FS Configuration Module is constructed from information provided by you in your User Parameter File, the same file used to create your AMX System Configuration Module.

The FS Configuration Module includes the **FS Parameter Table** which provides AMX/FS with the following information.

> Maximum number of tasks which can concurrently register for AMX/FS use
> Maximum number of open files
> Number of disk blocking buffers
> Device drivers required
> Logical drive allocation

The FS Configuration Module includes all of the private data storage that AMX/FS requires to accommodate the combination of user tasks, files, logical drives and device drivers which you have specified.

The FS Configuration Module also includes the following AMX/FS procedures:

Restart Procedure	`fj_restart()`
Exit Procedure	`fj_exit()`
Fault trap	`fjfsfault()`
Fatal trap	`fjfsfatal()`

2.2 AMX/FS Configuration Generation

The AMX Configuration Generator is a software generation tool which is used to create your AMX/FS Configuration Module from parameters which you provide in your User Parameter File, the same file used to create your AMX System Configuration Module. The User Parameter File contains a cryptic representation of your file system requirements.

The FS Configuration process is illustrated in the block diagram of Figure 2.2-1.

The AMX Configuration Manager (or a text editor) is used to create the User Parameter File which describes your AMX application requirements. This process is described in detail in Chapter 15 of your AMX User's Guide. Your User Parameter File must include the AMX configuration parameters defined in Chapter 2.4.

The User Parameter File must then be edited with the text editor of your choice to insert the AMX/FS keywords to describe your file system requirements. You then use the Configuration Generator to read your User Parameter File and produce the C source file called the FS Configuration Module.

The Configuration Generator uses a file called the FS Configuration Template as a model for your FS Configuration Module. This template file is merged with the information in your User Parameter File to produce your FS Configuration Module.

The C language FS Configuration Module must be compiled as described in the AMX/FS Tool Guide for inclusion in your AMX system.

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

Note

Your User Parameter File must include the AMX configuration parameters defined in Chapter 2.4.
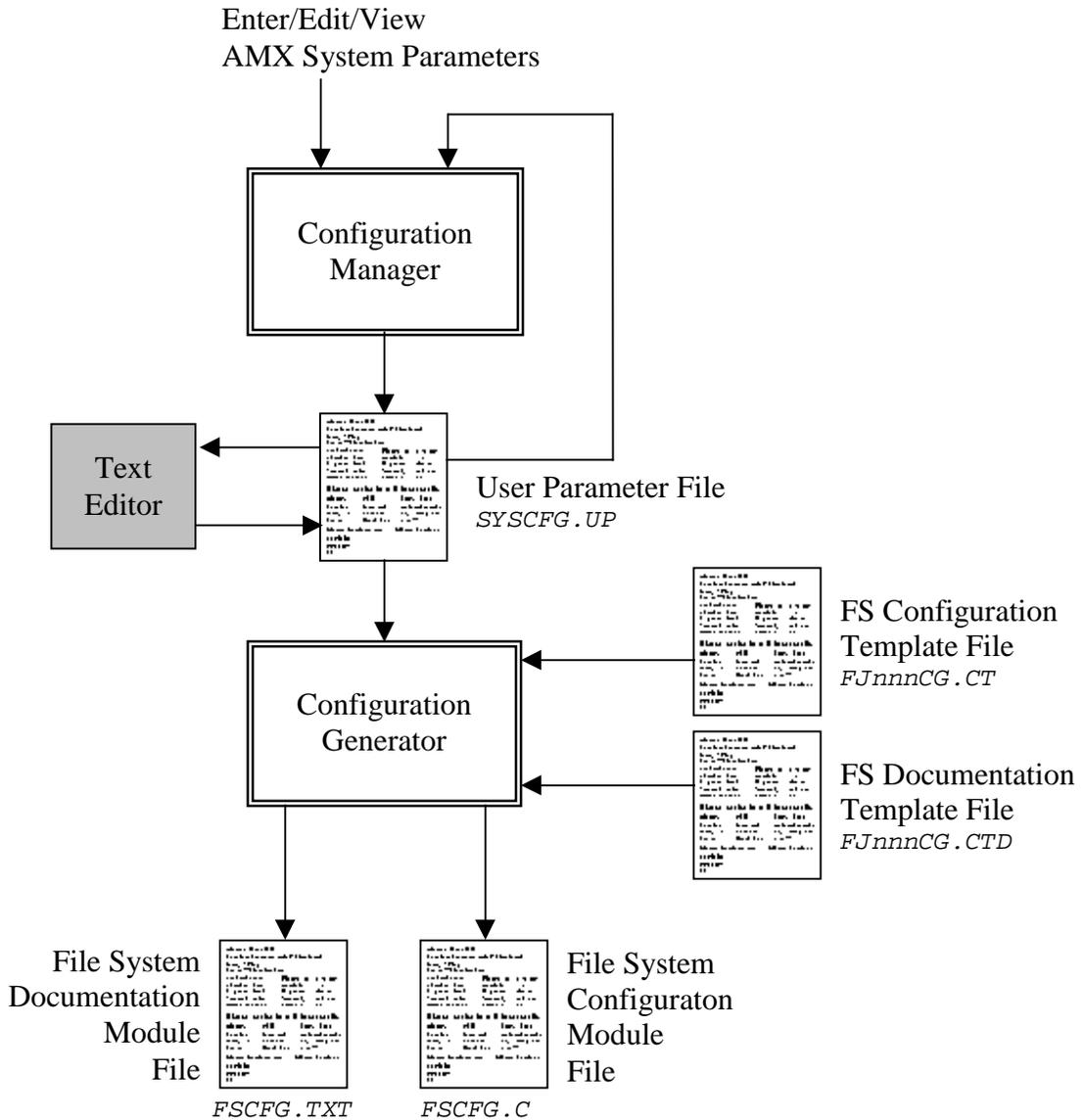
Figure 2.2-1 AMX/FS Configuration Building Process

Using the Generator

The AMX 86 Configuration Generator will operate on a PC or compatible running Microsoft® Windows® or compatible operating system.

The following files are required to create your FS Configuration Module. The Generator is delivered with AMX 86. The other files are provided with AMX/FS.

File	Purpose
AM831CG.EXE	AMX 86 Configuration Generator (utility program)
FJ838CG.CT	FS Configuration Template File
FJSAMSCF.UP	AMX/FS Sample Program User Parameter File

Using the AMX/FS Sample Program User Parameter File *FJSAMSCF.UP* as an example, take your own User Parameter File *SYSCFG.UP* and edit it to include a description of your AMX/FS disk requirements.

Copy files *AM831CG.EXE* and *FJ838CG.CT* and your file *SYSCFG.UP* into the directory in which you wish to build your FS Configuration Module. Make that directory the current directory, say *C:....*.

Start the Generator as follows.

```
C:....>AM831CG SYSCFG.UP FJ838CG.CT FSCFG.C
```

The Configuration Generator will read the User Parameter File *SYSCFG.UP* and merge it with the FS Configuration Template File *FJ838CG.CT* to produce FS Configuration Module *FSCFG.C*.

The Configuration Generator will also merge a file called the FS Documentation Template File *FJ838CG.CTD* with the information in your User Parameter File *SYSCFG.UP* to produce an FS Documentation Module, *FSCFG.TXT*, which is a text file summarizing the characteristics of your AMX/FS configuration.

```
C:....>AM831CG SYSCFG.UP FJ838CG.CTD FSCFG.TXT
```

The C language FS Configuration Module must be compiled as described in the toolset specific AMX Tool Guide for inclusion in your AMX system. The compiler will generate error messages which pin-point any inconsistencies in the file system parameters in your User Parameter File.

2.3 File System Parameters

The User Parameter File is a text file structured as illustrated in Figure 2.3-1. The file consists of a sequence of keywords of the form *...xxx* which begin in column one. Each keyword is followed by one or more parameters which you must provide.

```
; AMX/FS File System definitions
;
...FSYS              NUSERS,NBUFF,NFILES,REPORT
;
;
; AMX/FS RAM Disk Driver
;
...FSRAM             RDRIVE,RDNPAGE,RDSPAGE
;
;
; AMX/FS Floppy Driver
;
...FSFLP             SIZEA,SIZEB
;
;
; AMX/FS IDE Driver
;
...FSIDE             NLDRIVE
;
;
; AMX/FS PC BIOS Driver
;
...FSPCB             NLDRIVE
;
;
; AMX/FS User Device Driver (UDD)
;
...FSUDD             NLDRIVE,NFDRIVE,SHARE
;
;
; AMX/FS Logical Drive Table
;
...FSDRV             DNUM,DNAME,DNFAT
   :
   :
   :
...FSDRV             DNUM,DNAME,DNFAT
```

Figure 2.3-1 AMX/FS User Parameter File Entries

The example in Figure 2.3-1 uses symbolic names for all of the parameters following each of the keywords. The symbol names are replaced in your User Parameter File with the actual parameters needed in your system.

The keywords listed in Figure 2.3-1 must be present in the User Parameter File. With the exception of the *...FSDRV* keywords which make up the Logical Drive Table, the order of keywords in the User Parameter File is not particularly critical. The order of the keywords in Figure 2.3-1 may not match their order in the AMX/FS Sample Program User Parameter File *FJSAMSCF.UP* provided with AMX/FS.

Example 1:   Floppy Driver

```
; AMX/FS with:
; one 1.2 Mb floppy drive with no DMA restrictions
; 2 users, 10 blocking buffers, 5 open files, reporting enabled
;
...FSYS              2,10,5,1
...FSRAM
...FSFLP             1200,NONE
...FSIDE
...FSPCB
...FSUDD
;
;            AMX/FS Logical Drive Table
;
...FSDRV             0,floppya,9
```

Example 2:   IDE Driver and RAM Disk

```
; AMX/FS with:
; IDE driver
; 90 Kb RAM Disk
; 5 users, 20 blocking buffers, 10 open files, reporting disabled
;
...FSYS              5,20,10,0
...FSRAM             5,10,18
...FSFLP
...FSIDE             3
...FSPCB
...FSUDD
;
;            AMX/FS Logical Drive Table
;
...FSDRV             ~0,unuseda,0
...FSDRV             ~1,unusedb,0
...FSDRV             2,ide_C,64
...FSDRV             3,ide_D,64
...FSDRV             4,ide_E,64
...FSDRV             5,ram_drive,9
```

Example 3:   Floppy, IDE and UDD Drivers

```
; AMX/FS with:
; one 1.44 Mb floppy, one 1.2 Mb floppy,
; DMA buffer assigned by application procedure my_dmabuf()
; IDE drive with only one partition
; custom UDD driver with no "floppy" drives and
; two "hard" drives
; 4 users, 20 blocking buffers, 30 open files,
; reporting disabled by default
;
...FSYS              4,20,30
...FSRAM
...FSFLP             1440,1200
...FSIDE             1
...FSPCB
...FSUDD             2,0,0
;
;           AMX/FS Logical Drive Table
;
...FSDRV             0,floppy_A,9
...FSDRV             1,floppy_B,9
...FSDRV             2,ide_C,64
...FSDRV             3,udd_hd1,64
...FSDRV             4,udd_hd2,64
```

Example 4:   PC BIOS Driver and RAM Disk

```
; AMX/FS with:
; PC BIOS driver supporting floppy A and drives E and G
; 110 Kb RAM Disk with no page restrictions
; 5 users, 20 blocking buffers, 10 open files, reporting disabled
;
...FSYS              5,20,10,0
...FSRAM             7,1,220
...FSFLP
...FSIDE
...FSPCB             7
...FSUDD
;
;           AMX/FS Logical Drive Table
;
...FSDRV             0,floppya,9
...FSDRV             ~1,unusedb,0
...FSDRV             ~2,unusedc,0
...FSDRV             ~3,unusedd,0
...FSDRV             4,bios_E,64
...FSDRV             ~5,unusedf,0
...FSDRV             6,bios_G,64
...FSDRV             7,ram_drive,9
```

The User Parameter File MUST include a set of **file system definitions**.

```
...FSYS            NUSERS,NBUFF,NFILES,REPORT
```

       NUSERS   The maximum number of application tasks which can concurrently register to use AMX/FS.
Do not set *NUSERS* greater than the maximum number of tasks allowed in your AMX configuration.

       NBUFF   The number of disk blocking buffers that are available for traversing directories during file searches.
Each blocking buffer requires approximately 540 bytes.
A reasonable value for *NBUFF* is 20.
Increasing *NBUFF* improves performance if many tasks concurrently search for files.

       NFILES   The maximum number of files that can be open concurrently.
Each open file requires approximately 100 bytes.

       REPORT   Error reporting (logging) on a per task basis is:
0 = disabled (the default if *REPORT* is omitted)
1 = enabled

The User Parameter File MUST include one or more **device driver** specifications. The following list summarizes the AMX/FS device driver compatibilities.

Driver	Can be used with any of:
RAM Disk	All other device drivers
Floppy	RAM Disk, UDD[+], IDE driver
IDE	RAM Disk, UDD[+], Floppy driver
PC BIOS	RAM Disk, UDD[+]
UDD	RAM Disk, Floppy[+], IDE[+], PC BIOS drivers[+]

[+] as long as no device I/O access conflicts exist

Note

To omit a device driver, include the driver keyword
...FSxxx but omit all of the driver's parameters.

The **AMX/FS RAM Disk** is defined as follows.

```
...FSRAM          RDRIVE,RDNPAGE,RDSPAGE
```

        `RDRIVE`   Logical drive number (0 to n) allocated for the RAM Disk
                           in the AMX/FS Logical Drive Table

        `RDNPAGE`   Number of RAM Disk memory pages

        `RDSPAGE`   Size of each RAM Disk memory page
                           (measured in multiples of 512 bytes)

The RAM Disk's logical drive number MUST be the last logical drive in the AMX/FS Logical Drive Table.

The RAM Disk size is $RDNPAGE * RDSPAGE * 512$ bytes. For 16-bit segmented architectures, $RDSPAGE$ must be 64 or less giving a maximum page size of 32768 bytes. The maximum value for $RDNPAGE$ is then 32 giving a 1 Mb RAM Disk. Of course this absurd configuration leaves no memory free for your application. As a rule, set $RDNPAGE$ as small as possible and $RDSPAGE$ as large as necessary to achieve the desired RAM Disk size.

Storage for the RAM Disk will be allocated in the FS Configuration Module. This preconfigured RAM Disk must be initialized at run time by a task call to AMX/FS service procedure `fjfmtram()`.

You can alternatively dynamically allocate your RAM Disk. To do so, define $RDRIVE$ and $RDNPAGE$ as described above but set $RDSPAGE$ to `0`. You can then create your RAM Disk at run time using AMX/FS service procedure `fjfmtxram()`. The RAM Disk storage and the size of each page will be determined at that time by parameters in your call to `fjfmtxram()`.

Note: A maximum of sixteen (16) files, including directories, can reside in the root directory of the RAM Disk.

The **AMX/FS Floppy Driver** is defined as follows.

```
...FSFLP          SIZEA,SIZEB
```

        `SIZEA`   Size of floppy physical drive `A`
        `SIZEB`   Size of floppy physical drive `B`

        The allowable values for `SIZEA` and `SIZEB` are:
        `360`   Low density 5 1/4" 360 Kb drive
        `720`   Low density 3 1/2" 720 Kb drive
        `1200`   High density 5 1/4" 1.2 Mb drive
        `1440`   High density 3 1/2" 1.44 Mb drive
        `CMOS`   Drive size to be read from PC CMOS
        `NONE`   Drive is unused (not valid for drive `A`)

Floppy DMA Buffer

For AMX/FS 86 only, there is no special requirement to adjust the location or size of the floppy DMA buffer. For all other versions of AMX, the location and size of the floppy driver DMA buffer must be defined.

Remainder of this page left blank intentionally.

The **AMX/FS IDE Driver** is defined as follows.

```
...FSIDE          NLDRIVE
```

        NLDRIVE   Number of IDE logical drives

The IDE fixed disk controller can have one or two physical drives attached to it. Each physical drive can be subdivided into one or more partitions. By convention, these partitions correspond to the logical drives beginning with the drive id of *C*. If the first physical drive has two partitions, they will map to logical drives *C* and *D*. If a second physical drive is attached, its first partition will be drive *E* in this example.

Parameter *NLDRIVE* defines the number of logical drives, beginning with logical drive *C*, which the IDE driver must support. Only the first *NLDRIVE* logical drives present on the physical drives attached to the IDE controller will be accessible by the IDE driver.

The **AMX/FS PC BIOS Driver**, available only with AMX/FS 86, is defined as follows.

```
...FSPCB           NLDRIVE
```

        NLDRIVE   Number of PC BIOS logical drives

The PC BIOS supports a floppy disk controller with one or two physical floppy drives which map to logical drives *A* and *B*. The PC BIOS also supports a fixed disk controller with one or two physical fixed drives. By convention, the fixed drive partitions correspond to the logical drives beginning with the drive id of *C*.

Parameter *NLDRIVE* defines the number of PC BIOS logical drives, beginning with logical drive *A*, which the AMX/FS PC BIOS driver must support. Only the first *NLDRIVE* logical drives supported by the underlying PC BIOS will be accessible by the AMX/FS PC BIOS driver.

Note

The PC BIOS device driver is only available
with AMX/FS 86.

The **AMX/FS User Device Driver** is defined as follows.

```
...FSUDD          NLDRIVE,NFDRIVE,SHARE
```

     NLDRIVE   Number of UDD logical drives

     NFDRIVE   Number of UDD "floppy" drives

     SHARE   The UDD driver supports concurrent shared
access to its "floppy" drives and "hard" drives
(0 = no; 1 = yes)

The User Device Driver is a shell which can be customized to meet the requirements of your particular disk interface. The UDD will support two types of drives which, for convenience, are referred to as "floppy" drives and "hard" drives.

The UDD allows a total of *NLDRIVE* logical drives of which the first *NFDRIVE* can be "floppy" drives. The remaining *NLDRIVE* – *NFDRIVE* logical drives must correspond to "hard" drive partitions.

The drive id assigned to the first UDD logical drive is the first available drive id after all AMX/FS floppy driver, IDE driver and/or PC BIOS driver logical drive ids have been assigned. For example, if the IDE driver is used and has three logical drives (*C*, *D* and *E*), then drive *F* will be the UDD driver's first logical drive id.

The UDD logical drive ids are assigned first to the *NFDRIVE* "floppy" drives and then to the logical drives on the UDD's "hard" drives.

Since the UDD can support two different device types, it may be possible for the UDD to permit concurrent access by different tasks to each of the physical devices. This feature implies that the UDD is reentrant, and hence sharable concurrently by two tasks so long as each task requires access to a different physical device.

If your User Device Driver permits such shared access, set *SHARE* to *1*. Otherwise set *SHARE* to *0*. If in doubt, play it safe and set *SHARE* to *0*.

The **AMX/FS Logical Drive Table** is defined as follows.

```
...FSDRV          DNUM,DNAME,DNFAT

    DNUM          Logical drive number (0 based)

    DNAME         Name of logical drive's drive variable

    DNFAT         File Allocation Table buffer size
                  (measured in multiples of 512 bytes)
```

The AMX/FS Logical Drive Table must contain a description of each logical drive which AMX/FS must support. The entries in the table MUST be numerically ordered beginning with the entry for logical drive 0.

If a logical drive is not to be used, set its logical drive number to the complement of the drive number. For example, if floppy drive *B* is not used, its entry in the Logical Drive Table will be:

```
...FSDRV          ~1,unusedb,0            ; Floppy drive B unused
```

AMX/FS creates a drive variable for each logical drive defined in the Logical Drive Table. The drive variable is a public variable of type *int*. If the logical drive is used (*DNUM >= 0*), AMX/FS initializes the drive variable to contain the drive number. If the logical drive is unused (*DNUM < 0*), AMX/FS initializes the drive variable to contain the integer value *-1*.

The name of each drive variable must be provided by you. The name can be any name acceptable to C as a variable identifier. Each drive variable name must be unique.

Drive FAT Storage

AMX/FS maintains a copy in memory of the File Allocation Table (FAT) for each logical drive. For floppy disks, the size of the FAT for a particular drive depends on the media installed in the drive. For fixed disks, the size of the FAT for a particular drive depends on the size of the disk partition to which the logical drive corresponds.

The size of a FAT is measured in multiples of the 512 byte sector block size. The following list summarizes the FAT size for several different types of floppy disks and fixed drive partitions.

Media	Size	DNFAT	FAT Size
Floppy	360 Kb	2	1024
Floppy	720 Kb	3	1536
Floppy	1.2 Mb	7	3584
Floppy	1.44 Mb	9	4608
IDE	20 Mb	44	22528 (example only)
IDE	80 Mb	88	45056 (example only)

Parameter *DNFAT* defines the size of the FAT buffer allocated by AMX/FS for a particular logical drive. *DNFAT* is measured in multiples of 512 byte sector block size. If the FAT size specified by *DNFAT* matches the drive's actual FAT size, then AMX/FS will be able to maintain a copy of the drive's FAT in memory at all times and drive access performance will be at its best. If the FAT size specified by *DNFAT* is less than the drive's actual FAT size, then AMX/FS will swap FAT information in and out of the restricted FAT buffer as required. If the FAT size specified by *DNFAT* exceeds the drive's actual FAT size, then memory space will be wasted.

Although the FAT size for a large hard disk partition can be up to 256 blocks (128 Kb), AMX/FS will never try to use more than 64 blocks (32 Kb) at a time for FAT access. Therefore, there is no practical advantage to increasing *DNFAT* beyond 64.

If a logical drive is declared to be unused, set its FAT size parameter *DNFAT* to 0 to conserve memory space.

Note

If the FAT size specified by *DNFAT* is less than the drive's actual FAT size, then AMX/FS will swap FAT information in and out of the restricted FAT buffer as required.

2.4 AMX Configuration Requirements

In order to use the AMX/FS File System with AMX, you must update your User Parameter File to include parameters which define for AMX the set of resources which AMX/FS will require.

Restart and Exit Procedures

Add AMX/FS **Restart Procedure** `fj_restart()` to your list of application Restart Procedures. The position of `fj_restart()` in your list of Restart Procedures is not particularly important since no AMX/FS services can be used by your Restart Procedures.

Add AMX/FS **Exit Procedure** `fj_exit()` to your list of application Exit Procedures. The position of `fj_exit()` in your list of Exit Procedures IS important. Once AMX has called `fj_exit()` during its shutdown sequence, you cannot use any AMX/FS services. Therefore, you may find it best to place `fj_exit()` at the end of your Exit Procedure list. Of course, if you launch AMX for permanent execution, there is no need to include any reference to `fj_exit()` in your AMX configuration.

Time/Date Manager

The AMX **Time/Date Manager** is a prerequisite for the AMX/FS File System. AMX/FS uses its calendar time and date to time stamp files which are created or modified. In addition to including the Time/Date Manager in your configuration, you must ensure that the proper time and date are set before its first access by AMX/FS. This requirement will be met if you set the time and date before any drive is opened (mounted) for use.

If you cannot use the AMX Time/Date Manager for some reason, you will have to provide a replacement for AMX procedure `ajtdg()` or modify internal AMX/FS procedure `pc_getsysdate()` in module `FJ838XK.C`.

Timers

If you use the AMX/FS Floppy Driver, you must allocate one extra timer for its use. Simply increase the maximum number of AMX timers by one. The timer is used by the Floppy Driver to permit the floppy drive motor to be shut off after an approximate 3 second delay when the floppy drive is no longer in use.

Semaphores

AMX/FS requires a potentially large number of AMX semaphores for its proper operation. The maximum number of AMX semaphores provided in your AMX configuration must be increased by `NS4` which is computed as follows.

> `NS4 = 1 +` ((total number of logical drives) `* 3`)
> Increment `NS4` by one if the AMX/FS Floppy Driver is used.
> Increment `NS4` by one if the AMX/FS IDE Driver is used.

This page left blank intentionally.

3. AMX/FS Services

3.1 Introduction

The AMX/FS Library provides a wide range of file services from which to choose. Many of the services are optional and, if not used, will not even be present in your final system.

This section of the AMX/FS User's Guide differs from the others because it is intended for use as a programming guide. The remainder of this chapter introduces you to the AMX/FS programming environment. Chapter 3.3 provides descriptions of all of the procedures which are available in the AMX/FS Library in alphabetic order for easy reference.

All of the AMX/FS procedures are described using the C programming language. It is therefore recommended that you be at least superficially familiar with C.

A functional list of procedures is presented in Chapter 3.2. It is recommended that you use that chapter in conjunction with the procedure descriptions in Chapter 3.3 as follows. If you remember a procedure name but require information concerning its operation, go straight to the procedure description in the alphabetic list of procedures in Chapter 3.3. If you know what you wish to do functionally, but cannot remember the procedure name, go to Chapter 3.2 to quickly locate the procedure and then proceed to Chapter 3.3 to find its detailed description.

3.2 Summary of Services

The AMX/FS Library provides a wide range of file services from which to choose. Many of the services are optional and, if not used, will not even be present in your final system.

All of AMX/FS and its managers are fully reentrant and may be placed in Read Only Memory (ROM).

The following lists summarize all of the AMX/FS procedures which are accessible to the user. They are grouped functionally for easy reference.

Procedures are described in Chapter 3.3.

File System Control Services (class _fs_)

fjfserrfn	Install a task Error Handler
fjfserrno	Fetch AMX/FS error code
fjfsfatal	AMX/FS fatal trap
fjfsfault	AMX/FS fault trap
fjfsperror	Record (log) an AMX/FS error
fjfssignin	Register to use AMX/FS
fjfssignout	Renounce use of AMX/FS

Drive Access Services (class _drv_)

fjdrvabort	Abort all operations on a drive
fjdrvclose	Close (unmount) a disk drive
fjdrvget	Fetch task's current drive number
fjdrvgetcwd	Fetch task's current working directory
fjdrvnfree	Determine free space on a drive
fjdrvopen	Open (mount) a disk drive
fjdrvparse	Parse a drive name
fjdrvset	Set task's default drive

Directory Access Services (class _xx_ = none)

fjchdir	Change current working directory
fjmkdir	Make a directory
fjrmdir	Remove (delete) a directory

File Access Services (class *xx* = none)

fjchsize	Change size of (grow or shrink) a file
fjclose	Close a file
fjeof	Test if file pointer is at end of file
fjfattr	Get or set file attributes
fjffdone	Finished a file find search
fjfffirst	Find first occurrence of a matching file
fjffnext	Find next occurrence of a matching file
fjflush	Flush a file to disk
fjfstat	Fetch status of an open file
fjlseek	Move current file pointer
fjopen	Open a file
fjread	Read from a file
fjremove	Remove a file (synonym for *fjunlink*)
fjrename	Rename a file or directory
fjstat	Fetch status of a named file
fjtell	Fetch current file pointer
fjunlink	Unlink (delete) a file
fjwrite	Write to a file

Format Services (class *fmt*)

fjfmtfloppy	Format a floppy diskette
fjfmtram	Format (initialize) a preconfigured RAM Disk
fjfmtxram	Create and format (initialize) a RAM Disk

Miscellaneous Services (class *xx* = none)

fjisdir	Check if path is a directory
fjisvol	Check if path is a volume (drive)
fjmkfs	Make a file system on a drive
fjmkpath	Concatenate a pathname and filename

String Format Services (class *_xx* = none)

fj_itoa	Format an *int* as a string
fj_ltoa	Format a *long* as a string
fj_sprintf	Format a string
fj_stoa	Format a *short int* as a string
fj_strjust	Copy and justify a string

3.3 AMX/FS Procedures

A description of every AMX/FS Library procedure is provided in this chapter. The descriptions are ordered alphabetically for easy reference.

Italics are used to distinguish programming examples. Procedure names and variable names which appear in narrative text are also displayed in italics. Occasionally a lower case procedure name or variable name may appear capitalized if it occurs as the first word in a sentence.

Vertical ellipses are used in program examples to indicate that a portion of the program code is missing. Most frequently this will occur in examples where fragments of application dependent code are missing.

```
        :
        : /* Examine file status */
        :
```

Capitals are used for all defined AMX/FS filenames, constants and error codes. All AMX/FS procedure, structure and constant names can be readily identified according to the nomenclature introduced in Chapter 1.3.

A consistent style has been adopted for each description. The procedure name is presented at the extreme top right and left as in a dictionary. This method of presentation has been chosen to make it easy to find procedures since they are ordered alphabetically.

Purpose   A one-line statement of purpose is always provided.

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

> This block is used to indicate which of your AMX application procedures can call the AMX/FS procedure. The term ISP refers to the Interrupt Handler of a conforming ISP. A filled in box indicates that the procedure is allowed to call the AMX/FS procedure. In the above example, only tasks and Exit Procedures would be allowed to call the procedure.

Setup   The prototype of the AMX/FS procedure is shown.
The AMX/FS header file in which the prototype is located is identified.
Include AMX/FS header file *FJZZZ.H* for compilation.

> File *FJZZZ.H* is a generic AMX/FS include file which automatically includes the correct subset of the AMX and AMX/FS header files for a particular target processor. If you include *FJZZZ.H* instead of its KADAK part numbered counterpart (*FJnnn.H*), your AMX application source modules will be readily portable to other processors without editing.

> Note that file *FJZZZ.H* includes the generic AMX header files *AMX831CF.H*, *AMX831EC.H* and *AMX831SD.H* so there is no need to include them in your files. However, you can always include the *AMX831xx.H* header files before or after file *FJZZZ.H* without error.

Description   Defines all input parameters to the procedure and expands upon the purpose or method if required.

Interrupts   AMX/FS procedures must occasionally deal with the processor interrupt mask. The effect of each AMX/FS procedure on the interrupt state is NOT specified in this manual. The intervals, if any, are brief and occur only when manipulating I/O devices.

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

Most AMX/FS procedures return an integer error status. A positive value indicates success and a negative value indicates that an error occurred.

Error status (the equivalent of C's *errno*) can be recovered by calling procedure *fjfserrno()*. Note that error status is NOT kept in a single variable but is maintained on a per task basis so that each task using AMX/FS can determine the reason for its own errors.

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

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

For example, all AMX/FS procedures assume that an integer or unsigned integer is a 16 or 32-bit value dependent only upon the basic register width of the target processor.

Task Switch   Task switching effects are NOT described in this manual. Tasks using AMX/FS will, from time to time, be blocked waiting for device operations to complete or for resources which AMX/FS requires to satisfy the caller's request.

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

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

This page left blank intentionally.

Purpose   **Change Current Working Directory**

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

Setup   Prototype is in file *FJnnnKF.H*.
#include "FJZZZ.H"
*int CJ_CCPP fjchdir(const char *path);*

Description *path* is a path string which references the directory of interest. The
directory path may be ambiguous in which case the default drive and
the current working directory will be used to resolve the directory
reference.

If *path* contains a drivename, the current working directory is changed
for that drive without changing the default drive. Otherwise, the
current working directory is changed for the default drive.

Returns   Error status is returned.
   0   Call successful
   -1   Call failed.

Errors returned by *fjfserrno()*:
   FJ_ENOENT   Directory not found

Restrictions None

Example   *#include "FJZZZ.H"*

if (fjchdir("D:\\USR\\DATA\\FINANCE") < 0)
  fjfsperror("Cannot change working directory\n");

See Also   *fjisdir, fjdrvgetcwd*

Purpose   **Change Size of (Grow or Shrink) a File**

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

Setup   Prototype is in file *FJnnnKF.H*.
                 #include "FJZZZ.H"
                 int CJ_CCPP fjchsize(int handle, long newsize);

Description *handle* is the file handle of an open file.

                 newsize is the required file size in bytes. The file pointer is left at the
                 new end of file.

                 Change the size of the open file referenced by *handle* to *newsize*
                 bytes.

                 If *newsize* is less than the current file size, the file is shrunk. Any
                 characters in the file beyond *newsize* bytes are lost.

                 If *newsize* is greater than the current file size, the file is grown. The
                 file is filled with bytes of value *0* from the current end of file up to the
                 new end of file.

Returns   Error status is returned.
                 0   Call successful
                 -1   Call failed.

                 Errors returned by *fjfserrno()*:
                 FJ_EBADF   Invalid handle or open for read only
                 FJ_ENOSPC   I/O error occurred
                 FJ_EINVAL   *Newsize* is invalid
                                        You cannot shrink a file to a negative size.
                 FJ_ESHARE   The file is open with another handle.
                                        The size of a shared file cannot be changed.

Restrictions This procedure will use more than 530 bytes of the calling task's stack.

Example   *#include "FJZZZ.H"*

                 int   handle;

                 handle = fjopen("DATA.FIL", FJ_RDWR, 0);
                 if(handle != -1)
                 *   fjchsize(handle, 1024L);*

See Also   *fjopen*

Purpose   **Close a File**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjclose(int handle);
```

Description   *handle* is the file handle of an open file.

Close the file and update the disk by flushing the file and its directory entry to disk. Free all resources associated with *handle*.

Returns   Error status is returned.

0	Call successful
-1	Call failed.

Errors returned by *fjfserrno()*:

FJ_EBADF	Invalid file handle
FJ_ENOSPC	I/O error occurred

Restrictions   None

Example
```
#include "FJZZZ.H"

int      handle;

if(fjclose(handle) < 0)
  fjfsperror("Error closing file\n");
```

See Also   *fjopen, fjflush*

Purpose   **Abort All Operations on a Drive**

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

Setup   Prototype is in file *FJnnnKF.H.*
                 #include "FJZZZ.H"
                 *void CJ_CCPP fjdrvabort(const char *path);*

Description *path* is a drivename or fullpath.

                 If a task senses that there are problems with a drive, it should call
                 fjdrvabort(). All resources associated with that drive will be freed
                 but no disk writes will occur. All file handles associated with the drive
                 become invalid. After correcting the problem, the task can call
                 fjdrvopen() to re-mount the disk and open its files again.

Returns   Nothing

Restrictions None

Note   All tasks which have opened files on the drive will find that their file
                 handles are no longer valid.

Example   *#include "FJZZZ.H"*

                 *int problem_exists(char *drivenamep);*

                 if (problem_exists("A:")) {
                 * fjdrvabort("A:");*

                 * if (!problem_exists("A:"))*
                 *   fjdrvopen("A:");*
                 }

See Also   *fjdrvopen*

Purpose   **Close (Unmount) a Disk Drive**

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

Setup   Prototype is in file *FJnnnKF.H*.
                  #include "FJZZZ.H"
                  *int CJ_CCPP fjdrvclose(const char *path);*

Description   *path* is a drivename or fullpath.

                  Flush the File Allocation Table to disk and free all resources associated
                  with the drive.

Returns   Error status is returned.
                  0   Call successful
                  -1   Call failed.
                                        Path does not contain a valid drivename or
                                        the drive is outside the range of supported drives.

                  Errors returned by *fjfserrno()*:
                     None

Restrictions None

Example   *#include "FJZZZ.H"*

                  fjdrvclose("A:");

See Also   *fjdrvabort, fjdrvopen*

Purpose	**Fetch Task's Current Drive Number**

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

Setup   Prototype is in file *FJnnnKF.H*.
#include "FJZZZ.H"
int CJ_CCPP fjdrvget(void);

Description   Fetch the default drive number for the calling task.

Returns   The default drive number is returned.
   >=0   Call successful
   -1   The calling task is not a registered user.

Restrictions   None

Example   *#include "FJZZZ.H"*

   int   driveno;

   driveno = fjdrvget();

See Also   *fjdrvgetcwd, fjdrvset*

Purpose   **Fetch Task's Current Working Directory**

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

Setup   Prototype is in file *FJnnnKF.H*.
                 #include "FJZZZ.H"
                 *int CJ_CCPP fjdrvgetcwd(int driveno, char *bufp, int length);*

Description *driveno* is the drive number of interest to the caller. Set *driveno* to
                 FJ_DDRIVE to select the task's default drive.

                 bufp is a pointer to storage for a string copy of the rootpath which defines
                 the current working directory for drive *driveno*.

                 length is the size of the available storage referenced by *bufp*. The buffer
                 must be large enough to accommodate the return rootpath. Note that
                 the largest path string permitted by AMX/FS is *FJ_MAXPATH* bytes.

Returns   Error status is returned.
                 0   Call successful
                 -1   Call failed.
                                 Invalid drive number,
                                 buffer is too small or
                                 the calling task is not a registered user.

                 Errors returned by *fjfserrno()*:
                     None

Restrictions None

Example   *#include "FJZZZ.H"*

                 char   cwd[FJ_MAXPATH];

                 if (fjdrvgetcwd(FJ_DDRIVE, cwd, sizeof(cwd)) < 0)
                 * fjfsperror(*
                 * "Cannot get default drive current working directory\n");*

See Also   *fjdrvget, fjchdir*

Purpose	**Determine Free Space on a Drive**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
long CJ_CCPP fjdrvnfree(const char *path);
```

Description *path* is a drivename or fullpath.

Determine the number of unused bytes of disk storage which are available on a particular drive.

Returns   The number of free bytes on the drive.

>=0	Call successful
-1L	Call failed.
	Path does not contain a valid drivename or
	the drive is outside the range of supported drives.

Errors returned by *fjfserrno()*:
None

Restrictions None

Note   The first time *fjdrvnfree()* is called after a drive has been mounted, it must scan the drive's entire File Allocation Table to calculate the available free space. This operation may take a long time. Subsequent calls to *fjdrvnfree()* will incur no such penalty.

Example
```
#include "FJZZZ.H"

char      buf[80];
long      nfree;

nfree = fjdrvnfree("A:");
if (nfree >= 0)
  fj_sprintf(buf, "%ld bytes free on drive A:", nfree);
```

Purpose   **Open (Mount) a Drive**

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

Setup   Prototype is in file *FJnnnKF.H*.
                 #include "FJZZZ.H"
                 *int CJ_CCPP fjdrvopen(const char *path);*

Description *path* is a drivename or fullpath.

                 If not already open, the specified drive is opened for use by all tasks.
                 Block zero information is read from the drive to determine the drive's
                 characteristics and to confirm that the drive contains a valid MS-DOS
                 file system.

Returns   Error status is returned.
                    0   Call successful
                   -1   Call failed.
                                  Path does not contain a valid drivename or
                                  the drive is outside the range of supported drives.

                 Errors returned by *fjfserrno()*:
                   None

Restrictions None

Note   At least one task must open a drive before the drive can be accessed by
                 any task. There is no limit to the number of times that a drive can be
                 opened. Opening an open drive has no effect.

Example   *#include "FJZZZ.H"*

                 fjdrvopen("C:");

See Also   *fjdrvabort, fjdrvclose*

Purpose   **Parse a Drive Name**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjdrvparse(const char *path);
```

Description *path* is a drivename, fullpath or filepath.

The drive name in the *path* string is decoded and the corresponding drive number is returned to the caller. If *path* does not include a drivename, the default drive number for the calling task is returned.

Returns   The drive number is returned.
>=0	Call successful
-1	Drivename in *path* is not valid or
	the drive is outside the range of supported drives.

Restrictions None

Example
```
#include "FJZZZ.H"

extern char *getdrivename(void);
char    buf[80];
int     driveno;

driveno = fjdrvparse(getdrivename());
if (driveno < 0)
  fj_sprintf(buf, "Cannot get drive name");
```

See Also   *fjmkpath, fjdrvset*

Purpose   **Set Task's Default Drive**

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

Setup   Prototype is in file *FJnnnKF.H*.
                 #include "FJZZZ.H"
                 int CJ_CCPP fjdrvset(int driveno);

Description *driveno* is the drive number which the calling task wishes to establish as
                 its default drive.

                 If *driveno* is the value *FJ_DDRIVE*, the task's default drive number will
                 be left unaltered.

Returns   The total number of logical drives is returned.
                 >0   Call successful
                 -1   Call failed. *Driveno* is outside the range of supported
                             drives or the calling task is not a registered user.

                 Errors returned by *fjfserrno()*:
                     None

Restrictions None

Note   You can use this procedure to determine the number of logical drives
                 which have been defined in the Drive Allocation Table in your AMX/FS
                 Configuration Module.

Example   *#include "FJZZZ.H"*

                 *extern char *getdrivename(void);*
                 char   buf[80];
                 int   ndrive, driveno;

                 ndrive = fjdrvset(FJ_DDRIVE);
                 if (ndrive < 0)
                 * fj_sprintf(buf, "Cannot determine number of drives");*

                 driveno = fjdrvparse(getdrivename());
                 if (driveno < 0)
                 * fj_sprintf(buf, "Cannot get drive name");*
                 else if (driveno >= ndrive)
                 * fj_sprintf(buf, "Drive number is %d; %d drives exist",*
                 *   driveno, ndrive);*
                 else if (fjdrvset(driveno) < 0)
                 * fj_sprintf(buf, "Cannot set default drive %d", driveno);*

See Also   *fjdrvget*

Purpose	**Test if File Pointer is at End of File**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjeof(int handle);
```

Description   *handle* is the file handle of an open file.

Returns   Error status is returned.

1	Call successful (at end of file)
0	Call successful (not at end of file)
-1	Call failed.

Errors returned by *fjfserrno()*:
FJ_EBADF	Invalid handle

Restrictions   None

Example
```
#include "FJZZZ.H"

int       fpos;

if ( (fpos = fjeof(handle)) < 0)
   fjfsperror("Cannot test end of file\n");

else if (fpos == 0)
   fj_sprintf(buf, "Not at end of file\n");
```

See Also   *fjlseek, fjtell*

Purpose   **Get or Set File Attributes**

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

Setup   Prototype is in file *FJnnnKF.H.*
                #include "FJZZZ.H"
                *int CJ_CCPP fjfattr(const char *filename, int attrib);*

Description *filename* is a filepath identifying the file of interest. The filepath may be
                ambiguous in which case the default drive and the current working
                directory will be used to resolve the file reference.

                attrib is the new attributes for the file. Set *attrib* to *-1* to read the file's
                attributes without modifying them. To modify the file attributes, set
                attrib to the bit-wise OR of the following bit masks which are defined
                symbolically in AMX/FS header file *FJnnnKF.H.*

                FJ_DA_NORMAL   Normal file; read and write access
                FJ_DA_RDONLY   Read only file
                                  File of same name cannot be created
                FJ_DA_HIDDEN   Hidden file; directory search will miss
                FJ_DA_SYSTEM   System file; directory search will miss
                FJ_DA_VOLUME   File is a volume
                FJ_DA_DIRENT   File is a directory
                FJ_DA_ARCHIVE   Archive; set when file changes

Returns   File attributes are returned.
                != -1   Call successful
                -1   Call failed.

                Errors returned by *fjfserrno()*:
                FJ_ENOENT   Path or file not found
                FJ_EACCESS   Access not permitted

Restrictions You MUST NOT try to set the *FJ_DA_VOLUME* or *FJ_DA_DIRENT* attributes

Note   The file is opened and then closed. The file is not open upon return.

Example   *#include "FJZZZ.H"*

                if (fjfattr("C:\\MYDIR\\FILE.X", -1) < 0)
                * fjfsperror("Cannot fetch file attributes\n");*

See Also   *fjfstat, fjopen, fjstat, fjunlink*

Purpose	**Finished a File Find Search**

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

Setup   Prototype is in file *FJnnnKF.H*.
                #include "FJZZZ.H"
                *void CJ_CCPP fjffdone(struct fjxffind *searchp);*

Description *searchp* is a pointer to a file searching structure used by AMX/FS to scan
                a directory looking for files with names matching a specific pattern.
                The search structure *fjxffind* is defined in header file *FJnnnKF.H*.

                A file matching search is initiated with a call to *fjfffirst()* and
                continued with zero or more calls to *fjffnext()*. When the search is
                complete, procedure *fjffdone()* must be called to permit AMX/FS to
                free all of the private resources which it had in use during the search.

Returns   Nothing

Restrictions Do not call *fjffdone()* unless the search structure referenced by *searchp*
                has first been initialized by a successful call to *fjfffirst()*.

Example   See *fjffnext()* example.

See Also   *fjfffirst, fjffnext*

Purpose	**Find First Occurrence of a Matching File**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjfffirst(struct fjxffind *searchp,
                      const char *filename);
```

Description   *searchp* is a pointer to a file searching structure used by AMX/FS to scan
                  a directory looking for files with names matching a specific pattern.
                  The search structure *fjxffind* is defined in header file *FJnnnKF.H*.

                  filename is a filepath which determines the directory in which searches
                  will take place. The filename component of the filepath is a file name
                  pattern which AMX/FS must use to find matching files. The filebase
                  and extension can contain the DOS wildcard characters * and ?.

                  AMX/FS uses the search structure to maintain its copy of the fullpath
                  and file name pattern which it derives from *filename*. It also uses the
                  search structure to monitor its progress during the search process.

Returns   Error status is returned.
>   *1*   Call successful (Matching file found)
>   *0*   Call failed (No matching file)

                  Errors returned by *fjfserrno()*:
                      Undefined

                  If successful, the search structure will contain information describing the
                  matching file.

Restrictions Once the search structure referenced by *searchp* has been initialized by a
                  successful call to *fjfffirst()*, it must not be altered by the task other
                  than through subsequent calls to *fjffnext()* to continue the search.

                  The directory being searched will be locked for exclusive access by the
                  calling task. The directory will remain locked by that task until its search
                  is complete and the search structure is "closed" with a call to *fjffdone()*.

Note   This procedure does NOT open the first matching file which it finds.

Example   See *fjffnext()* example.

See Also   *fjffdone, fjffnext*

Purpose   **Find Next Occurrence of a Matching File**

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

Setup   Prototype is in file *FJnnnKF.H.*
                 #include "FJZZZ.H"
                 *int CJ_CCPP fjffnext(struct fjxffind *searchp);*

Description *searchp* is a pointer to a file searching structure used by AMX/FS to scan
                 a directory looking for files with names matching a specific pattern.
                 The search structure *fjxffind* is defined in header file *FJnnnKF.H.*

                 AMX/FS continues the directory scan which was initiated by an earlier
                 call to *fjfffirst()* referencing the same search structure.

Returns   Error status is returned.
1	Call successful (Matching file found)
0	Call failed (No matching file)

                 Errors returned by *fjfserrno()*:
                     Undefined

                 If successful, the search structure will contain information describing the
                 matching file.

Restrictions Procedure *fjffnext()* must NOT be called unless the search structure
                 referenced by *searchp* has been initialized by a successful call to
                 fjfffirst(). The search structure must not be altered by the task other
                 than through subsequent calls to *fjffnext()* to continue the search.

                 The directory being searched remains locked for exclusive access by the
                 calling task. The directory will remain locked by that task until its search
                 is complete and the search structure is "closed" with a call to *fjffdone()*.
                 If the call to *fjffnext()* fails, the search structure must be "closed" with
                 a call to *fjffdone()*.

Note   This procedure does NOT open the next matching file which it finds.

Example
```
#include "FJZZZ.H"

extern void recordfiles(char *msgp);

char     buf[80];
struct fjxffind search;

if ( fjfffirst(&search, "A:\\dev\\*.c") ) {
  do {
          /* Record filebase, extension and size        */
          fj_sprintf(buf, "%-8s.%-3s%7ld\n",
                    search.xffname,
                    search.xffext,
                    search.xffsize);

          recordfiles(buf);

          } while ( fjffnext(&search) );

  /* Call fjffdone to declare the search over          */
  fjffdone(&search);
  }
```

See Also   *fjffdone, fjfffirst*

Purpose   **Flush a File to Disk**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjflush(int handle);
```

Description *handle* is the file handle of an open file.

The file's data, if any, and directory entry are written to the disk. The
drive's File Allocation Table is also transferred to disk. After this
procedure completes, the on-disk view of the file matches the in-
memory view. It is good practice to periodically flush a file to disk if
the file is being extended. If a file is not flushed or closed and a power
failure or drive failure occurs, the disk copy of the file will be incorrect
and the integrity of the disk's File Allocation Table will be
compromised.

Returns   Error status is returned.
> *0*   Call successful
> *-1*   Call failed.

Errors returned by *fjfserrno()*:
> *FJ_EBADF*   Invalid file handle
> *FJ_ENOSPC*   I/O error occurred

Restrictions None

Example
```
#include "FJZZZ.H"

int      handle;

if ( fjflush(handle) < 0)
  fjfsperror("Error flushing file\n");
```

See Also   *fjclose*

Purpose   **Format a Floppy Diskette**

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

Setup   Prototype is in file *FJnnnKF.H.*
                  #include "FJZZZ.H"
                  int CJ_CCPP fjfmtfloppy(int driveno, int disksize,
                  *   int interleave, FJ_CALLBACK callback);*

Description   *driveno* is the drive number (*0* or *1*) of the disk drive containing the
                  floppy diskette to be formatted. The diskette must be present in the
                  drive and the drive door must be closed.

                  disksize is the size of the floppy diskette. It is not the size of the drive in
                  which the floppy has been inserted. *Disksize* must be one of the
                  following:

360	Low density 5 1/4" 360 Kb diskette
720	Low density 3 1/2" 720 Kb diskette
1200	High density 5 1/4" 1.2 Mb diskette
1440	High density 3 1/2" 1.44 Mb diskette

                  interleave is the interleave factor which determines the order in which
                  the sectors on each track are recorded. If *interleave* is *1*, sectors will
                  be recorded sequentially which may yield the highest possible transfer
                  rate. However, if the floppy controller, the floppy DMA controller and
                  the memory system cannot sustain sequential sector transfers, an
                  interleave of *1* will yield the worst possible transfer rate requiring
                  one complete diskette revolution per sector.

                  callback is a pointer to an application procedure which will be called as
                  each sector is formatted allowing the application to monitor the
                  formatting process. Set *callback* to *(FJ_CALLBACK)CJ_NULLFN* if
                  such a call back is not required. The call back procedure is prototyped
                  as follows:

                  void CJ_CCPP callback(int track, int ntrack);

                  track is the number of the track being formatted numbered from *1* to
                  ntrack.

                  ntrack is the total number of tracks on the diskette.

Returns   Error status is returned.

0	Call successful
-1	Call failed.

Errors returned by *fjfserrno()*:
Undefined

Restrictions This procedure is only available with the optional AMX/FS Floppy Driver.

Example
```
#include "FJZZZ.H"

extern int flp_B;              /* Floppy B: drive variable  */
void showstring(char *bufp);

void CJ_CCPP showtracks(int track, int ntrack)
{
  char    buf[80];

  fj_sprintf(buf, "Formatting track %d of %d.",
                  track, ntrack);
  showstring(buf);
  }

void CJ_CCPP task(void)
{
  fjfmtfloppy(flp_B, 1200, 3,
                  (FJ_CALLBACK)showtracks);
}
```

See Also   *fjmkfs*

Purpose   **Format (Initialize) a Preconfigured RAM Disk**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjfmtram(int driveno);
```

Description   *Driveno* is the logical drive number assigned to the RAM Disk in your AMX/FS Configuration Module.

The memory assigned for use as a RAM Disk is prepared for use and a valid file system is transferred to the "disk" making it accessible as a valid drive.

Returns   Error status is returned.

0	Call successful
-1	Call failed.
	Driveno does not match the drive number
	configured for use as a RAM Disk.

Errors returned by *fjfserrno()*:
   None

Restrictions The RAM Disk must be formatted before any task attempts to open (mount) the drive. The RAM Disk must only be formatted once by one task.

Warning   Upon return from *fjfmtram()*, the calling task is no longer registered as an AMX/FS user even if the task had been previously registered.

Example
```
#include "FJZZZ.H"

extern int ram_disk;          /* RAM Disk drive variable   */

fjfmtram(ram_disk);
```

See Also   *fjmkfs, fjfmtxram*

Purpose   **Create and Format (Initialize) a RAM Disk**

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

Setup   Prototype is in file *FJnnnKF.H.*
                #include "FJZZZ.H"
                *int CJ_CCPP fjfmtxram(int driveno, char CJ_CCHUGE *ramp,*
                                          int npage, unsigned int spage);

Description *driveno* is the logical drive number assigned to the RAM Disk in your
                AMX/FS Configuration Module.

                ramp is a pointer to a region of memory to be used as storage for the RAM
                    Disk. For AMX 86 users, the storage must be a huge array to permit it
                    to exceed 64 Kb in total.

                npage is the number of pages into which the RAM Disk is to be
                    subdivided. This number must be less than or equal to that provided in
                    the RAM Disk definition in your AMX/FS Configuration Module.

                spage is the size of each RAM Disk page measured in multiples of 512
                    bytes.

                The RAM Disk size is *npage * spage * 512*. There must be at least
                this amount of storage available at the memory location referenced by
                ramp. See the RAM Disk configuration description in Chapter 2.3 for a
                more detailed definition of *npage* and *spage*.

                The memory assigned for use as a RAM Disk is prepared for use and a
                valid file system is transferred to the "disk" making it accessible as a
                valid drive.

Returns   Error status is returned.
                0   Call successful
                -1   Call failed.
                                 Driveno does not match the drive number
                                 configured for use as a RAM Disk or
                                 npage exceeds that defined for your RAM Disk
                                 in your FS Configuration Module.

                Errors returned by *fjfserrno()*:
                    None

Restrictions   The RAM Disk must be formatted before any task attempts to open (mount) the drive. The RAM Disk must only be formatted once by one task.

Warning   Upon return from *fjfmtxram( )*, the calling task is no longer registered as an AMX/FS user even if the task had been previously registered.

Example
```
#include "FJZZZ.H"

extern int ram_disk;           /* RAM Disk drive variable   */

#define NPAGE 8
#define SPAGE 64

static char ramdisk[NPAGE * SPAGE * 512];

fjfmtxram(ram_disk, ramdisk, NPAGE, SPAGE);
```

See Also   *fjmkfs, fjfmtram*

Purpose	**Install a Task Error Handler**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
void CJ_CCPP fjfserrfn(FJ_CALLBACK errorfnp);
```

Description *errorfnp* is a pointer to the calling task's Error Handler. The Error Handler is described in Chapter 1.6.

Pointer *errorfnp* is installed in the task's User Access Block for future reference by AMX/FS when the task tries to report errors using procedure *fjfsperror()*.

If *errorfnp* is *(FJ_CALLBACK)CJ_NULLFN*, the task's current Error Handler is cancelled causing subsequent calls by the task to *fjfsperror()* to have no effect.

Returns   Nothing

Restrictions If the calling task is not a registered AMX/FS user, the installation of the Error Handler will fail with no notification back to the caller.

Example   See the example provided in Chapter 1.6.

See Also   *fjfsperror*

Purpose   **Fetch a Task's Error Number**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjfserrno(void);
```

Description   AMX/FS returns the error number recorded in the calling task's User Access Block. The error number indicates the result of the most recent file operation performed by AMX/FS in response to a request from the calling task.

Returns   Error number is returned.

0	Last operation successful (no error recorded)
>0	Last call failed with this error code
FJ_ENOUSER	Calling task is not a registered user.

Restrictions   None

Example
```
#include "FJZZZ.H"

if (fjfserrno())
  fjfsperror("Last AMX/FS operation failed\n");
```

See Also   *fjfsperror*

Purpose   **AMX/FS Fault and Fatal Error Traps**

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

Setup   Prototype is in file *FJnnnKF.H.*
                 #include "FJZZZ.H"
                 void CJ_CCPP fjfsfatal(int error);
                 void CJ_CCPP fjfsfault(int error);

Description *error* is an internal AMX/FS error code defining a serious fault or a fatal
                 condition. These error codes are used to identify errors listed in file
                 FJnnnERR.C.

                 Procedures *fjfsfatal()* and *fjfsfault()* are NOT to be called by
                 the application. They are called only by AMX/FS. Prior to calling
                 either of these procedures, AMX/FS will first try to report the fault by
                 calling *fjfsperror()* to allow the application task which produced the
                 fault to record the reason.

                 These procedures reside in the AMX/FS Configuration Module and are
                 provided for application testing purposes.   You should place
                 breakpoints on these procedures to detect serious problems at the
                 earliest possible opportunity.

                 Calls by AMX/FS to *fjfsfault()* are serious internal faults which
                 will always be reported to the application as a failed operation. They
                 usually occur because of misuse of AMX/FS by the application. The
                 faults, although serious, are always recoverable within AMX/FS itself
                 but the application which produced the fault probably will need
                 correction.

                 Calls by AMX/FS to *fjfsfatal()* are catastrophic faults which, if
                 ignored, will lead to the failure of AMX/FS and possibly to eventual
                 corruption of your disk content. AMX/FS hangs in an infinite loop in
                 the context of the task which generated the fatal condition.

Returns   *fjfsfault*   Nothing
                 fjfsfatal   Never

Restrictions None

See Also   *fjfserrno, fjfsperror*

Purpose	**Record (Log) an Error Condition**
Used by	■ Task □ ISP □ Timer Procedure □ Restart Procedure ■ Exit Procedure
Setup	Prototype is in file *FJnnnKF.H*. *#include "FJZZZ.H"* *void CJ_CCPP fjfsperror(char *msgp);*

Description *msgp* is a pointer to an application message string or *CJ_NULL* if none is required.

AMX/FS examines the calling task's error number. If the task is not a registered user or if its most recently recorded error number is *0*, the request is ignored.

If the task calling *fjfsperror()* has NOT installed an Error Handler, the request is ignored even if an error has been recorded.

If the calling task has installed an Error Handler (see *fjfserrfn()*), AMX/FS calls the handler, passing it the message pointer *msgp*, if provided by the caller, and one or more message strings describing the recorded error.

Returns Nothing

Restrictions None

Example See the example provided in Chapter 1.6.

See Also *fjfserrno, fjfserrfn*

Purpose   **Register to Use AMX/FS**
                  Renounce Use of AMX/FS

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

Setup   Prototype is in file *FJnnnKF.H*.
                  ```
                  #include "FJZZZ.H"
                  int CJ_CCPP fjfssignin(void);
                  void CJ_CCPP fjfssignout(void);
                  ```

Description   Any task wishing to use AMX/FS must first register (sign in) as an
                  AMX/FS user. AMX/FS assigns a User Access Block to the task for
                  subsequent use by AMX/FS as it services file access requests by the
                  task. Although AMX/FS will automatically register a task if necessary,
                  a task can explicitly register itself by calling *fjfssignin()*.

                  A registered task which no longer requires access to the file system can
                  renounce use of AMX/FS by calling *fjfssignout()*.

Returns   *fjfssignout*   Nothing
                  fjfssignin   Returns error status.
                  0   Call successful
                  -1   Call failed.
                                         No User Access Blocks are available.

                  Errors returned by *fjfserrno()*:
                      None

Restrictions A registered task MUST not be deleted. The task must renounce use of
                  AMX/FS before it permits itself to be deleted.

Note   Task registration remains in effect until the task renounces its use of
                  AMX/FS. A task procedure which ends execution remains registered.
                  The task will still be registered when it next executes.

Example   ```
                  #include "FJZZZ.H"

                  if (fjfssignin() == 0) {
                    :
                    : Use file system
                    :
                    fjfssignout();
                  }
                  ```

**Purpose**      **Fetch Status of an Open File**

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

**Setup**        Prototype is in file *FJnnnKF.H.*
                 *#include "FJZZZ.H"*
                 *int CJ_CCPP fjfstat(int handle, struct fjxstat *statp);*

**Description**  *handle* is the file handle of an open file.

                 *statp* is a pointer to storage for the file's status information.  The structure
                     *fjxstat* is defined in AMX/FS header file *FJnnnKF.H.*

**Returns**      Error status is returned.
                 *0*              Call successful
                 *-1*             Call failed.

                 Errors returned by *fjfserrno()*:
                 *FJ_EBADF*       Invalid file handle
                 *FJ_ENOSPC*      I/O error occurred

**Restrictions**  None

**Note**         The file remains open.

**Example**      *#include "FJZZZ.H"*

                 *int      handle;*
                 *struct fjxstat filestatus;*

                 *if ( fjfstat(handle, &filestatus) < 0)*
                 *  fjfsperror("Cannot fetch file status using handle\n");*

**See Also**     *fjfattr, fjopen, fjstat*

| | |
|---|---|
| **Purpose** | **Test if Path is a Directory or Volume** |

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

**Setup**     Prototype is in file *FJnnnKF.H.*
```
#include "FJZZZ.H"
int CJ_CCPP fjisdir(const char *path);
int CJ_CCPP fjisvol(const char *path);
```

**Description** *path* is a path string.  The path may be ambiguous in which case the
default drive and the current working directory will be used to resolve
the directory reference.

*fjisdir* checks the path by opening and then closing the directory to
determine if a valid directory exists.

*fjisvol* checks the path by opening the drive to determine if a valid
drive exists.  The drive remains open.

**Returns**   Error status is returned.

| | |
|---|---|
| *1* | Path is a directory (volume) |
| *0* | Path is NOT a directory (volume) |

Errors returned by *fjfserrno()*:

| | |
|---|---|
| *FJ_ENOENT* | Directory not found |

**Restrictions** None

**Example**   
```
#include "FJZZZ.H"

char *mypath = "F:\\MYDIR";
char   buf[80];
void showstring(char *bufp);

if (fjisvol(mypath)) {
  if (fjisdir(mypath)) {
        fj_sprintf(buf, "%s is a valid path.", mydir);
        showstring(buf);
        }
    }
```

**See Also**  *fjchdir, fjmkdir*

**Purpose**       **Move Current File Pointer**

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

**Setup**         Prototype is in file *FJnnnKF.H*.
                  *#include "FJZZZ.H"*
                  *long CJ_CCPP fjlseek(int handle, long offset, int where);*

**Description**   *handle* is the file handle of an open file.

                  *offset* is the distance which the file pointer must be moved relative to the
                      origin determined by parameter *where*.

                  *where* is the origin relative to which the file pointer will be moved.
                      *Where* must be one of the following:

                  | *FJ_SEEK_SET* | Seek from the beginning of the file |
                  | *FJ_SEEK_CUR* | Seek from the current file pointer |
                  | *FJ_SEEK_END* | Seek from the end of the file |

                  Attempting to seek beyond the end of the file leaves the file pointer at
                  the end of the file. Any attempt to position the file pointer to a negative
                  displacement in the file is rejected as an error.

**Returns**       The resulting file pointer position is returned.
                  | *>=0* | Call successful |
                  | *-1L* | Call failed. |

                  Errors returned by *fjfserrno()*:
                  | *FJ_EBADF* | Invalid file handle |
                  | *FJ_EINVAL* | Seek to negative displacement in file |

**Restrictions**  None. If the call fails, the file pointer is left unaltered.

**Example**       *#include "FJZZZ.H"*

                  *char      buf[80];*
                  *long      record, rec_number, rec_size;*

                  *rec_number = 80L;*
                  *rec_size = 120L;*
                  *record = rec_number * rec_size;*
                  *if ( fjlseek(handle, record, FJ_SEEK_SET) != record) {*
                  *  fj_sprintf(buf, "Cannot find record %ld\n", rec_number);*
                  *  fjfsperror(buf);*
                  *  }*

**See Also**      *fjeof, fjtell*

**Purpose**    **Make a Directory**

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

**Setup**    Prototype is in file *FJnnnKF.H*.
*#include "FJZZZ.H"*
*int CJ_CCPP fjmkdir(const char *path);*

**Description**    *path* is a path string identifying the new directory to be created. The path
may be ambiguous in which case the default drive and the current
working directory will be used to resolve the directory reference.

The drivename and all of the subdirectory dirnames in *path* which are
required to resolve the final location of the new directory must be valid.
The final dirname in *path* must not exist as a file or directory.

**Returns**    Error status is returned.
| | |
|---|---|
| *0* | Call successful |
| *-1* | Call failed. |

Errors returned by *fjfserrno()*:
| | |
|---|---|
| *FJ_ENOENT* | Directory not found |
| *FJ_EEXIST* | File or directory already exists |
| *FJ_ENOSPC* | Write failed |

**Restrictions**    None

**Example**    *#include "FJZZZ.H"*

*if ( fjmkdir("\\USE\\LIB\\HEADER\\SYS") < 0)*
  *fjfsperror("Cannot make directory\n");*

**See Also**    *fjisdir, fjrename, fjrmdir*

**Purpose**     **Make a File System**

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

**Setup**     Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fjmkfs(int driveno, struct fjxformat *fmtp)
```

**Description**     *driveno* is the drive number of the disk drive to be formatted with an MS-DOS file system.

        *fmtp* is a pointer to a structure which defines the characteristics of the disk drive to be formatted. Typical values for different drive types are shown in Appendix A. The structure *fjxformat* is defined in AMX/FS header file *FJnnnKF.H*.

        If the entry *fmtp->xfmsecpfat* is set to zero, the number of sectors required for the File Allocation Table will be calculated by AMX/FS from other information in the structure.

**Returns**     Error status is returned.
        *0*                  Call successful
        *-1*                Call failed.

    Errors returned by *fjfserrno()*:
        Undefined

**Restrictions**     The disk drive must be low level formatted and, if necessary, partitioned before *fjmkfs* can format it with an MS-DOS file system.

**Note**     The calling task will be forced to wait until it can gain exclusive use of the drive which is to be formatted. Upon return, the drive is closed (unmounted) and must be opened (mounted) again before it can be used by any tasks.

---

### Warning

ALL DATA on the drive formatted by *fjmkfs* will be lost!

---

**Example**

```
#include "FJZZZ.H"

extern int ide_E;              /* IDE drive variable      */
struct fjxformat fmt;          /* Format parameter block  */

/* Format an 80 Mb hard disk partition                    */

strcpy(fmt.xftoemname,"KADAK");
fmt.xftdriveno = 0;            /* Physical disk 0 or 1     */
fmt.xftmedia = 0xF8;
fmt.xftsecpalloc = 8;
fmt.xftnumfats = 2;
fmt.xftsecrsv = 1;
fmt.xftsecpfat = 88;
fmt.xftnumroot = 512;
fmt.xftsecptrk = 17;
fmt.xftnumhead = 8;
fmt.xftnumcyl = 1224;
fmt.xftnlabel = 0x12345678L;
strcpy(fmt.xfttlabel, "AMX/FS v1.0");

if ( pc_mkfs(ide_E, &fmt) < 0)
   fjfsperror("File system format failed\n");
```

See Also   `fjfmtfloppy, fjfmtram, fjfmtxram`

Purpose	**Concatenate a Pathname and a Filename**

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

Setup   Prototype is in file `FJnnnKF.H`.
```
#include "FJZZZ.H"
char * CJ_CCPP fjmkpath(char *buf, const char *path,
                        const char *filename);
```

Description   `buf` is a pointer to storage which is large enough to hold the resulting filepath string.

`path` is a pathname string. The pathname string is considered valid with or without a final file separator character `\`.

`filename` is a valid filename consisting of a filebase and an option extension. No path information is provided with `filename`.

Returns   A pointer to `buf`.

Restrictions None

Example
```
#include "FJZZZ.H"

char    buf[FJ_MAXLENGTH];

fjmkpath(buf, "A:DIR", "FILE1");
/* buf is 'A:DIR\FILE1'                              */

fjmkpath(buf, "\\DIR\\DIR2\\", "FILE2.EXT");
/* buf is '\DIR\DIR2\FILE2.EXT'                      */
```

Purpose   **Open a File**

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

Setup   Prototype is in file *FJnnnKF.H*.

```
#include "FJZZZ.H"
int CJ_CCPP fjopen(const char *filename, unsigned int access,
                        unsigned int mode);
```

Description *filename* is a filepath string. The filepath may be ambiguous in which case the default drive and the current working directory will be used to resolve the file reference.

access defines the file access rights to be used in opening the file. The valid access values are determined by bit-wise OR of the following bit masks which are defined symbolically in AMX/FS header file *FJnnnKF.H*.

FJ_O_BINARY	Ignored (all file access is binary)
FJ_O_TEXT	Ignored (all file access is binary)
FJ_O_RDONLY	Open for read only
FJ_O_WRONLY	Open for write only
FJ_O_RDWR	Open for read and/or write access
FJ_O_CREAT	Create the file if it does not exist
	Has no effect if the file already exists
FJ_O_EXCL	Only used with *FJ_O_CREAT*
	Fail if the file exists; do not open
FJ_O_TRUNC	Truncate the file to zero length if the file exists
	No effect if the file does not exist
FJ_O_NOSHAREANY	Fail if already open
	Fail in future if another open is tried
FJ_O_NOSHAREWR	Fail if already open for write
	Fail in future if another open for write is tried

mode defines the allowable access modes to be given to the file if a new file is created. Set mode to *0* if *access* does not include *FJ_O_CREAT*. The valid access mode values are determined by bit-wise OR of the following bit masks which are defined symbolically in AMX/FS header file *FJnnnKF.H*.

FJ_S_IWRITE	Write permitted
FJ_S_IREAD	Read permitted (Always true)

Returns   A file handle is returned.

               `>=0`   Call successful
               `-1`   Call failed.

          Errors returned by `fjfserrno()`:

`FJ_ENOENT`	File or path to file not found
`FJ_EMFILE`	No file handles available (too many files open)
`FJ_EEXIST`	Create failed; file exists
	Open with `FJ_O_EXCL` failed; file exists
`FJ_EACCES`	Attempt to open a read only file for write
	or to open a drive or directory
`FJ_ENOSPC`	Create failed
`FJ_ESHARE`	Sharing error; file is already open

Restrictions None

Example

```
#include "FJZZZ.H"

int      handle;


handle = fjopen("\\USR\\MYFILE",
                FJ_O_CREAT | FJ_O_EXCL | FJ_O_WRONLY,
                FJ_S_IWRITE);
if (handle < 0)
  fjfsperror("Cannot create file\n");


handle = fjopen("\\USR\\MYFILE",
                FJ_O_RDWR | FJ_O_NOSHAREWR, 0);
if (handle < 0)
  fjfsperror("Cannot open file for non-sharable write\n");
```

See Also   `fjclose, fjread, fjwrite`

fjread   **fjread**

Purpose   **Read From a File**

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

Setup   Prototype is in file *FJnnnKF.H*.
                   #include "FJZZZ.H"
                   *int CJ_CCPP fjread(int handle, void *buf, unsigned int count);*

Description   *handle* is the file handle of an open file.

                   buf is a pointer to storage for the data read from the file.

                   count is the number of bytes (characters) to be transferred. The allowable
                       range for count is *1* to *~0 - 1* bytes.

Returns   The number of bytes read is returned.
                   0   Call successful (no bytes read)
                   n != -1   Call successful
                                      (unsigned int)n = number of bytes read
                   -1   Call failed.

                   Errors returned by *fjfserrno()*:
                   FJ_EBADF   Invalid file handle
                   FJ_ENSOPC   File I/O error

Restrictions None

Note   The file pointer is incremented by *n* bytes if the read is successful.

Example

```
#include "FJZZZ.H"

int        handle;
int        handle2
long       buff[512/sizeof(long)];
int        n;

handle = fjopen("FROM.FIL", FJ_O_RDONLY, 0);
handle2 = fjopen("TO.FIL", FJ_O_CREAT | FJ_WRONLY,
                              FJ_S_IWRITE)

if ( (handle >= 0) && (handle2 >= 0) ) {

   while(1) {
           n = fjread(handle, (char *)buff,
                             sizeof(buff));
           if (n == -1) {
                   fjfsperror("File read failed\n");
                   break;
                   }
           if (n == 0)
                   break;
           if (fjwrite(handle2, (char *)buff, n) != n)
                   fjfsperror("File write failed\n");
           }
     }
```

See Also   `fjwrite`

fjrename   **fjrename**

Purpose   **Rename a File or Directory**

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

Setup   Prototype is in file *FJnnnKF.H*.
                   #include "FJZZZ.H"
                   *int CJ_CCPP fjrename(const char *oldname, const char *newname);*

Description   *oldname* is a filepath identifying the location and current name of the file
                   to be renamed. The filepath may be ambiguous in which case the
                   default drive and the current working directory will be used to resolve
                   the file reference.

                   newname is the new filename to be given to the file referenced by
                   oldname. No path information is provided with *newname*.

                   The file at the location specified by *oldname* is renamed *newname*. If
                   the file specified by *oldname* does not exist, if *newname* is not a valid
                   filename or if a file with name *newname* already exists in the directory
                   specified by *oldname*, the rename will fail.

Returns   Error status is returned.
                   0   Call successful
                   -1   Call failed.

                   Errors returned by *fjfserrno()*:
                   FJ_ENOENT   *oldname* path or file not found
                   FJ_EEXIST   File *newname* already exists
                   FJ_ENOSPC   Directory update failed

Restrictions   None

Note   You can use *fjrename* to rename directories and volume labels.

Example   *#include "FJZZZ.H"*

                   if (fjrename("\\USR\\TXT\\LETTER.TXT", "LETTER.BAK") < 0)
                   * fjfsperror("Cannot rename LETTER.TXT\n");*

See Also   *fjmkdir, fjrmdir*

Purpose   **Remove (Delete) a Directory**

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

Setup   Prototype is in file *FJnnnKF.H.*
                 #include "FJZZZ.H"
                 *int CJ_CCPP fjrmdir(const char *path);*

Description *path* is a pathname string. The pathname may be ambiguous in which
                 case the default drive and the current working directory will be used to
                 resolve the directory reference.

                 The directory specified by *path* is deleted. If *path* does not reference a
                 directory, if the directory has read only access rights or if the directory
                 is not empty, the directory is not deleted.

Returns   Error status is returned.
                 0   Call successful
                 -1   Call failed.

                 Errors returned by *fjfserrno()*:
                 FJ_ENOENT   Directory not found
                 FJ_EACCES   Not a directory, not empty or in use
                 FJ_ENOSPC   Write failed

Restrictions You cannot delete a directory which is not empty or which is currently in
                 use through an active file handle.

Note   You cannot delete a file using *fjrmdir*. Use *fjunlink* for that purpose.

Example   *#include "FJZZZ.H"*

                 if (fjrmdir("D:\\USR\\TEMP") < 0)
                 * fjfsperror("Cannot delete directory\n");*

See Also   *fjmkdir, fjrename, fjunlink*

Purpose   **Fetch Status of a File**

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

Setup   Prototype is in file *FJnnnKF.H*.
                   ```
                   #include "FJZZZ.H"
                   int CJ_CCPP fjstat(const char *filename, struct fjxstat *statp);
                   ```

Description   *filename* is a filepath identifying the file of interest. The filepath may be
                   ambiguous in which case the default drive and the current working
                   directory will be used to resolve the file reference.

                   statp is a pointer to storage for the file's status information. The structure
                   fjxstat is defined in AMX/FS header file *FJnnnKF.H*.

Returns   Error status is returned.
                   0   Call successful
                   -1   Call failed.

                   Errors returned by *fjfserrno()*:
                   FJ_ENOSPC   I/O error occurred

Restrictions None

Note   The file is opened and then closed. The file is not open upon return.

Example   ```
                   #include "FJZZZ.H"

                   struct fjxstat filestatus;

                   if (fjstat("C:\\MYDIR\\FILE.X", &filestatus) < 0)
                     fjfsperror("Cannot fetch file status\n");
                   ```

See Also   *fjopen, fjfattr, fjfstat*

Purpose   **Fetch Current File Pointer**

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

Setup   Macro is in file *FJnnnKF.H*.
                  #include "FJZZZ.H"
                  #define fjtell(handle) fjlseek(handle, 0L, FJ_SEEK_CUR);

Description   *handle* is the file handle of an open file.

                  This procedure is implemented as a macro which uses procedure
                  fjlseek to move the file pointer *0* bytes from its current position.

Returns   The resulting file pointer position is returned.
                  >=0   Call successful
                  -1L   Call failed.

                  Errors returned by *fjfserrno()*:
                  FJ_EBADF   Invalid file handle

Restrictions None

Note   If the call fails, the file pointer is left unaltered.

Example   *#include "FJZZZ.H"*

                  char   buf[80];
                  long   record, rec_number, rec_size;

                  rec_number = 80L;
                  rec_size = 120L;
                  *record = rec_number * rec_size;*

                  if (fjlseek(handle, record, FJ_SEEK_SET) != record) {
                  * fj_sprintf(buf, "Cannot find record %ld\n", rec_number);*
                  * fjfsperror(buf);*
                  * }*

                  if (fjtell(handle) != record)
                  * fjfsperror("Cannot read file pointer\n");*

See Also   *fjeof, fjlseek*

Purpose   **Unlink (Delete) a File**

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

Setup   Prototype is in file `FJnnnKF.H`.
                   `#include "FJZZZ.H"`
                   `int CJ_CCPP fjunlink(const char *filename);`
                   `fjremove` is defined in file `FJnnnKF.H` to be `fjunlink`.

Description   `filename` is a filepath string. The filepath may be ambiguous in which
                   case the default drive and the current working directory will be used to
                   resolve the file reference.

                   The file specified by `filename` is deleted. If `filename` does not
                   reference a file, if the file has the read only attribute or if the file is
                   currently open, the file is not deleted.

Returns   Error status is returned.
                   `0`   Call successful
                   `-1`   Call failed.

                   Errors returned by `fjfserrno()`:
                   `FJ_ENOENT`   Path or file not found
                   `FJ_EACCES`   Cannot delete a directory, an open file
                                    or a read only file
                   `FJ_ENOSPC`   Write failed

Restrictions You cannot delete an open file or a file marked with the read only
                   attribute.

Note   You cannot delete a directory using `fjunlink`. Use `fjrmdir` for that
                   purpose.

Example   `#include "FJZZZ.H"`

                   `if( fjunlink("B:\\USR\\TEMP\\TMP001.PRN") < 0)`
                   `  fjfsperror("Cannot delete file\n");`

See Also   `fjfattr, fjrmdir`

Purpose   **Write to a File**

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

Setup   Prototype is in file *FJnnnKF.H*.
               #include "FJZZZ.H"
               *int CJ_CCPP fjwrite(int handle, void *buf, unsigned int count);*

Description   *handle* is the file handle of an open file.

               buf is a pointer to the data to be written to the file.

               count is the number of bytes (characters) to be transferred. The allowable
                  range for count is *1* to *~0 - 1* bytes.

Returns   The number of bytes written is returned.
               0   Call successful (no bytes written)
               n != -1   Call successful
                                (unsigned int)n = number of bytes written
               -1   Call failed.

               Errors returned by *fjfserrno()*:
               FJ_EBADF   Invalid file handle or open for read only
               FJ_ENOSPC   No space on disk or I/O error

Restrictions The file must have been opened for write access.

Note   The file pointer is incremented by *n* bytes if the write is successful.

Example   See *fjread()* example.

See Also   *fjread*

fj_itoa	**fj_itoa**
fj_ltoa	**fj_ltoa**
fj_stoa	**fj_stoa**

Purpose **Format Number as a String**

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

Setup Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
char * CJ_CCPP fj_itoa(int num, char *destp, int base);
char * CJ_CCPP fj_ltoa(long num, char *destp, int base);
char * CJ_CCPP fj_stoa(short int num, char *destp, int base);
```

Description *num* is an integer, long or short integer numeric value.

destp is a pointer to storage for the text string representation of the number *num*. The storage must be adequate to hold the longest string needed to represent the number *num*.

base is the base of the number system which is to be used in the conversion of num to a text string. Base can be *2*, *8*, *10* or *16*. Base is not checked.

Returns The pointer *destp* is returned.

A null terminated string representing the numeric value of *num* using the base *base* numbering system is stored at **destp*.

Restrictions None

Note Hexadecimal numbers (*base = 16*) are generated using lower case letters. The minus sign will only be present for negative values of *num* if the base is decimal (*base = 10*).

Example
```
#include "FJZZZ.H"

char      buf[20];

fj_itoa(3206, buf, 10);      /* buf = "3206"              */
fj_ltoa(65536L, buf, 16);    /* buf = "10000"             */

fj_stoa(-2, buf, 10);        /* buf = "-2"                */
fj_stoa(-2, buf, 16);        /* buf = "fffe"              */
```

See Also *fj_sprintf*

Purpose	**Format a String**

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

Setup   Prototype is in file *FJnnnKF.H*.
```
#include "FJZZZ.H"
int CJ_CCPP fj_sprintf(char *destp, const char *fmt, ...);
```

Description   *destp* is a pointer to storage for the text string specified by *fmt*. The storage must be adequate to hold the longest string needed to meet the format specified by *fmt*.

fmt is a pointer to a format specification string. This string is copied to the destination buffer with conversion specifications in the string, if any, translated to text string representations of the variables which follow *fmt*.

fj_sprintf is a reentrant, minimal version of the *sprintf* procedure found in most C run-time libraries. Refer to your C library reference manual for a description of *sprintf* (it may be described under *printf* or *fprintf*). The following features are supported.

-,+,' '	left justify and sign control
#	alternate *0x* and *0X* formats
w.p	width and precision
	(precision is allowed but is ignored)
h,l	short and long variable conversions
d,i,u	decimal integer variables
x,X	hexadecimal integer variables
c,s	character and string variables

The following features are NOT supported.

*	variables used as width specifiers
o	octal numbers
e,E,f,g,G	floating point variables or notation
n	length assignment to a variable
p,P	pointers

Returns   The number of characters transferred to the destination buffer.

Restrictions   None

Example   See *fjffnext()* example.

See Also   *fj_itoa, fj_ltoa, fj_stoa*

Purpose   **Copy and Justify a String**

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

Setup   Prototype is in file *FJnnnKF.H*.
                 #include "FJZZZ.H"
                 char * CJ_CCPP fj_strjust(char *destp, char *srcp,
                 *   int leftjust, int width, char padchar);*

Description *destp* is a pointer to storage for the text string specified by *srcp*. The
                 storage must be adequate to hold the longest string needed to justify
                 string *srcp* within a field of width determined by parameter *width*. If
                 the length of string *srcp* is *ls*, then the storage size must be the greater
                 of *ls+1* or *width+1*.

                 srcp is a pointer to the source string which will be copied to the
                 destination buffer.

                 leftjust is non-zero if the source string is to be left justified within the
                 destination buffer. If *leftjust* is zero, the source string will be right
                 justified within the destination buffer.

                 width is the width (that is length) of the resulting string stored in the
                 destination buffer. If *width* is less than the length of the source string,
                 then *width* is increased to match the length of the source string.

                 padchar is the pad character which will be used to pad the source string
                 on the left (*leftjust == 0*) or on the right (*leftjust != 0*).

Returns   The *destp* pointer to the destination buffer.

Restrictions None

Example   *#include "FJZZZ.H"*

                 char   buf[20];

                 fj_strjust(buf, "Left", 8, 1, '+');
                 /* buf = "Left++++"   */

                 fj_strjust(buf, "Right", 8, 0, '-');
                 /* buf = "---Right"   */

                 fj_strjust(buf, "Too right", 8, 0, '-');
                 /* buf = "Too right"   */

4. AMX/FS Drivers

4.1 Preconfigured Drivers

The AMX/FS File System includes a preconfigured RAM Disk Driver and PC BIOS Driver. The latter is only provided with AMX/FS 86 for AMX 86 users.

RAM Disk Driver

The RAM Disk Driver is provided in the AMX/FS Library. It is ready for use. It does not have to be customized. To use it in your application, include a RAM Disk driver definition in your User Parameter File as described in Chapter 2.3.

The RAM Disk must be initialized at some time after your AMX application has been launched. The RAM Disk must only be initialized once. It must be initialized by a single task before any task can open (mount) the RAM Disk logical drive for use.

If you have preconfigured your RAM Disk so that its memory is allocated within the FS Configuration Module, then the task must call AMX/FS procedure `fjfmtram()` to initialize the RAM Disk.

If you have defined your RAM Disk to permit its memory to be dynamically allocated, then the task must call AMX/FS procedure `fjfmtxram()` to initialize the RAM Disk. In this case, the task must provide a pointer to the memory which is reserved for use as a RAM Disk and indicate the amount of storage so allocated. The memory region can be allocated by your task in any way that best suits your application.

Once initialized, the RAM Disk logical drive can be opened and manipulated just like any other drive.

Source code for the RAM Disk Driver is provided with AMX/FS in file `FJnnnRAM.C`. You may find that this driver is an excellent template for a custom device driver for any disk subsystem with characteristics similar to those of a RAM Disk.

Warning

Any task using the RAM Disk will be compute bound and will therefore prevent all lower priority tasks from executing.

The RAM Disk will be reformatted every time AMX is launched.

PC BIOS Driver

The PC BIOS Driver, available only with AMX/FS 86, is provided in the AMX/FS Library. It is ready for use. It does not have to be customized. To use it in your application, include a PC BIOS driver definition in your User Parameter File as described in Chapter 2.3.

The PC BIOS Driver supports a floppy disk controller with one or two physical floppy drives which map to logical drives *A* and *B*. The PC BIOS Driver also supports a fixed disk controller with one or two physical fixed drives. By convention, the fixed drive partitions correspond to the logical drives beginning with the drive id of *C*.

The PC BIOS Driver will automatically initialize itself with the first attempt by any task to open (mount) a logical drive which corresponds to a floppy drive or a hard disk partition. Of course, the logical drive must lie within the range of logical drives supported by the PC BIOS Driver as specified by you in your FS Configuration Module.

Since the PC BIOS is not reentrant, two tasks cannot concurrently perform disk operations on different logical drives serviced by the PC BIOS Driver. For example, one task cannot access a file on a floppy disk while another task is manipulating a file on a logical drive on a fixed disk. To do that, you will have to use the AMX/FS Floppy and IDE Drivers.

Source code for the PC BIOS Driver is provided with AMX/FS 86 in file *FJ838PCB.C*. You will find that this driver is an adaptation of the UDD custom device driver described in Chapter 4.2.

+---+
| Note |
| |
| The PC BIOS device driver is only available |
| with AMX/FS 86. |
+---+

4.2 Custom Drivers

A disk driver called a User Device Driver (UDD) is provided with AMX/FS. The User Device Driver is a shell which can be customized to meet the requirements of your particular disk interface. To use it in your application, include a UDD driver definition in your User Parameter File as described in Chapter 2.3.

The UDD will support two types of drives which, for convenience, have been referred to as "floppy" drives and "hard" drives. Within this chapter, these two drive types will be called F_drives and H_drives. F_drives are considered to use removable media and are not partitioned. H_drives are fixed drives with one or more partitions per H_drive.

Source code for the UDD is provided with AMX/FS in file *FJnnnUDD.C*. You must edit this file to meet your disk interface requirements. All sequences within file *FJnnnUDD.C* which require editing are marked as follows.

```
/* ===== Begin Modification ======= */
```

The UDD C source file must be compiled to produce an object module, *FJnnnUDD.OBJ* which must then be linked with your AMX application as described in the AMX/FS Tool Guide.

UDD Drives

The UDD will support *NF* physical F_drives and *NH* physical H_drives. These values are established by the definitions of symbols *NFDRIVE* and *NHDRIVE* in file *FJnnnUDD.C*. The UDD is initially set to support two F_drives (*NFDRIVE = 2*) and two H_drives (*NHDRIVE = 2*). Edit these symbol definitions to meet your needs.

Your UDD driver definition in your User Parameter File must match your definitions of *NFDRIVE* and *NHDRIVE* in file *FJnnnUDD.C*. The total number of logical drives supported by the UDD, *NLDRIVE*, will ultimately depend on the number of partitions on each of the H_drives. Let *NP* be the total number of partitions on the *NH* H_drives. Then, *NLDRIVE = NF + NP*. The following examples illustrate acceptable configurations of the UDD.

UDD File *FJnnnUDD.C*			User Parameter File	
NF	*NH*	*NP*	*NLDRIVE*	*NFDRIVE*
2	2	2	4	2
1	2	4	5	1
0	1	3	3	0
2	0	0	2	2
1	1	2	3	1

> **Warning**
>
> If the UDD driver definition in your User Parameter File does not match the definitions of *NFDRIVE* and *NHDRIVE* in file *FJnnnUDD.C*, you will be unable to open any of the UDD logical drives without error.

UDD Procedures

There are four procedures within the UDD which you must edit to meet your disk interface requirements.

udd_dvc_enter()	UDD driver entry
udd_dvc_init()	UDD device initialization
udd_dvc_io()	UDD device read or write
udd_dvc_exit()	UDD driver exit

The UDD driver entry procedure *udd_dvc_enter()* will be called upon the first attempt by any task to open (mount) any UDD logical drive. Of course, the logical drive must lie within the range of logical drives supported by the UDD as specified by you in your FS Configuration Module. Procedure *udd_dvc_enter()* must ready the UDD such that each of its F_drives and H_drives can be accessed. This procedure should initialize any semaphores or timers which it needs for its operation. It is at this point that you should install Interrupt Service Procedures, if any, required to service your devices.

Once the driver entry setup is complete, AMX/FS can proceed to open the particular UDD logical drive of interest. Whenever any UDD logical drive is to be opened, AMX/FS calls UDD procedure *fj_udd_open()* which in turn calls its device initialization procedure *udd_dvc_init()*. This procedure must identify the characteristics of the F_drive or H_drive of interest: number of heads, number of cylinders (tracks), number of sectors per track, etc.

The manner in which *udd_dvc_init()* determines the drive characteristics is up to you. The default UDD driver simply fetches the information from a data table embedded in the driver. You may be able to interrogate your disk controller to derive the information.

Once a logical drive has been opened, AMX/FS will be able to read data from and write data to the corresponding F_drive or H_drive. It does so by calling UDD procedure *fj_udd_io()* which in turn calls its device I/O procedure *udd_dvc_io()*. The procedure is instructed to read or write a particular number of sectors at a particular location (head, cylinder, starting sector) on the F_drive or H_drive.

The default UDD driver illustrates the process of translating the parameters received by *udd_dvc_io()* into the typical values needed by a disk controller.

The final UDD procedure which requires your attention is the driver exit procedure *udd_dvc_exit()*. This procedure will only be called by AMX/FS if your AMX application terminates through an orderly AMX shutdown and exit. This procedure is called by AMX/FS Exit Procedure *fj_exit()* as it attempts to shut down the file system. All operations on all F_drives and H_drives will have been completed or terminated by AMX/FS before this procedure is invoked.

Your procedure *udd_dvc_exit()* must place all F_drives and H_drives into an idle state, disable the controllers from generating further interrupts and, if applicable, restore any interrupt vectors which the UDD used.

4.3 Floppy Driver

The AMX/FS Floppy Driver, available as an option, is provided separate from the AMX/FS Library. To use it in your application, include a floppy driver definition in your User Parameter File as described in Chapter 2.3.

The Floppy Driver supports a NEC 765 class of floppy disk controller with an Intel 8237 DMA controller used for data transfer between the floppy controller and memory. The floppy controller can have one or two physical floppy drives attached to it. These disk drives map to logical drives *A* and *B*.

The Floppy Driver supports 360 Kb, 760 Kb, 1.2 Mb and 1.44 Mb disk drives. Diskettes of these densities can be used in any drive which supports the particular media. For example, 360 Kb diskettes can be used in some 1.2 Mb drives.

Since the two floppy disk drives share a single controller, two tasks cannot concurrently perform disk operations on separate floppy drives. However, one floppy drive can be accessed by one task while a second task references another drive such as a RAM Disk, an IDE drive or a custom UDD drive.

The Floppy Driver will automatically initialize itself with the first attempt by any task to open (mount) a logical drive which corresponds to a floppy drive. During its initialization, the Floppy Driver resets the floppy controller and installs an Interrupt Service Procedure for the floppy controller interrupt. The driver creates an AMX semaphore which it uses to signal completion or termination of data transfers. It also creates an AMX timer which is used to control the drive motor shut off sequence. Your AMX System Configuration Module must provide enough timers and semaphores to permit one of each to be used by the Floppy Driver.

The Floppy Driver is ready for use with AMX 86 in 80x86 environments with a PC compatible floppy controller and DMA interface. When used on non-PC platforms, the Floppy Driver will require porting as described in this chapter.

Source code for the Floppy Driver is provided separate from the AMX/FS File System in the following files.

FJnnnFLP.C	Floppy Driver AMX/FS interface and control
FJnnnFLB.C	AMX kernel interface
FJnnnFLC.C	Floppy and DMA controller interface

In addition to these modules, several board support procedures are provided by the AMX/FS File System in file *FJnnnBRD.ASM*.

The Floppy Driver C source files and the AMX/FS board support module must be compiled and assembled to produce object modules *FJnnnFLP.OBJ*, *FJnnnFLB.OBJ*, *FJnnnFLC.OBJ* and *FJnnnBRD.OBJ*. These object modules must then be linked with your AMX application. This procedure is described in the AMX/FS Tool Guide.

Formatting Diskettes

The AMX/FS service procedure `fjfmtfloppy()`, described in Chapter 3.3, can be used by any task to format a floppy diskette for use with AMX/FS. This procedure is located in module `FJnnnFLP.C`. If you have no need to format floppy diskettes, you can delete procedure `fjfmtfloppy()` and its private counterpart `fj_flp_format()` from module `FJnnnFLP.C`, thereby conserving some code space.

Porting Issues - General

When used on non-PC platforms, the Floppy Driver will require porting. Porting the driver for use with any NEC 765 class of floppy disk controller and Intel 8237 DMA controller will be simple. Porting for use with a different DMA controller should not be difficult. However, to port the driver for use with a different floppy controller may require significant change.

To port the Floppy Driver for use in non-PC environments, you must edit the Floppy Driver files to meet your floppy and DMA controller interface requirements. All sequences within files which require editing are marked as follows.

```
/* ===== Begin Modification ======= */
```

In file `FJnnnFLP.C`, adjust the definitions of the various timeout constants to match your drive specifications. By default the data transfer completion timeout is set to 6 seconds and the motor shutoff timeout is 3 seconds. Other time constants are floppy controller and drive specific.

In file `FJnnnFLC.C`, modify the definitions of the floppy controller and DMA controller I/O port numbers or device addresses. You may also have to modify the definition of the input and output macros which are used to transfer 8-bit values to and from external hardware devices.

Most non-PC platforms will not have a CMOS register bank which defines the types (densities) of the attached floppy disk drives. To eliminate this feature, alter the definition of symbol `_FL_CMOS` in file `FJnnnFLC.C` to be `0`.

The NEC 765 floppy controller requires a 25 microsecond delay between accesses to its main status register. To accommodate the requirement on today's high speed processors, the Floppy Driver uses procedure `fj_flp_delay25()` in file `FJnnnFLC.C` to enforce the delay. By default, the driver uses board support procedure `fj_brd_tick()` to monitor the instantaneous downcount value of the PC's Intel 8253 timer chip until a 25 microsecond delay (measured with 0.8 microsecond resolution) has been achieved.

You will have to modify procedures `fj_flp_delay25()` in file `FJnnnFLC.C` and/or `fj_brd_tick()` in file `FJnnnBRD.ASM` to meet this requirement. If you have no such timing resolution hardware available, you can delete `fj_brd_tick()` and modify to `fj_flp_delay25()` to sit in a tight software loop for at least 25 microseconds. Note that such a solution will depend on the processor clock frequency and will therefore not be portable to all versions of your target processor.

Porting Issues - Interrupts

The AMX/FS Floppy Driver includes an Interrupt Service Procedure ready for use with the particular version of AMX which you are using.

The AMX/FS 86 Floppy Driver builds its own ISP root and installs the pointer to that root into the processor's Interrupt Vector Table.The ISP Root provides access to the Floppy Driver's Interrupt Service Procedure `fj_flp_isr()` in module `FJnnnFLB.C`.

In file `FJnnnFLC.C`, you must modify the definition of the floppy controller's interrupt vector number. If necessary, modify the Floppy Driver support procedures in module `FJnnnBRD.ASM`.

Porting Issues - DMA Controller

There are two procedures in file `FJnnnFLC.C` which may have be modified to accommodate the manner in which your DMA controller is programmed. Procedure `fj_flp_dmainit()` must set up the DMA controller in preparation for a DMA transfer by the floppy controller. The default Floppy Driver prepares channel 2 of an Intel 8237 DMA Interrupt Controller for the transfer.

Procedure `fj_flp_dma()` includes a short sequence in which the terminal count status of the DMA controller is tested. The default Floppy Driver reads the status register of the Intel 8237 DMA Interrupt Controller and tests the channel 2 terminal count status bit.

Porting Issues - DMA Buffering

The segmented architecture of the Intel 80x86 family of processors introduces a DMA buffering complexity that is rarely encountered with other processor architectures. It is also possible that memory addressing limitations of a DMA controller or its hardware interface implementation may also introduce DMA buffering restrictions.

The data transfer limitations are of two types.

In some cases, a DMA transfer is not allowed to straddle an absolute 64 Kb physical segment boundary. That is, sequential access from any physical address `0xXXXXFFFF` to `0xYYYY0000` where `YYYY` is `XXXX+1` is prohibited.

In other cases, the allowed range of memory address accessible to the DMA controller may be restricted. For example, the PC's Intel 8237 DMA Interrupt Controller is only able to access the first one megabyte of physical memory.

The AMX/FS Floppy Driver accommodates these DMA buffering restrictions. The DMA buffering issues involved in porting the driver are processor dependent as described below.

The default **AMX/FS 86 Floppy Driver** will require no modification if your DMA controller can access the entire one megabyte of the real mode 80x86 address space with or without physical 64 Kb segmentation restrictions. If you plan to use the AMX/FS 86 Floppy Driver with a DMA controller with restricted memory access, you will have to increase the definition of symbol *DMA_BUFSIZE* in file *FJnnnFLC.C* beyond its default value of *1* and, if necessary, arrange that the private DMA buffer *fj_flp_dmabuf[]* declared in that module lies entirely within a region of memory accessible by your DMA controller. Instructions are provided in the file.

4.4 IDE Driver

The AMX/FS IDE Driver, available as an option, is provided separate from the AMX/FS Library. To use it in your application, include an IDE driver definition in your User Parameter File as described in Chapter 2.3.

The IDE Driver supports any disk controller which meets the industry standard IDE (integrated drive electronics) specification. The IDE controller can have one or two physical fixed disk drives attached to it. By convention, these disk drives map to logical drives beginning with logical drive C.

Since the two fixed disk drives share a single controller, two tasks cannot concurrently perform disk operations on any of the logical drives present on the fixed disks. However, one logical drive can be accessed by one task while a second task references another drive such as a RAM Disk, a floppy drive or a custom UDD drive.

The IDE Driver will automatically initialize itself with the first attempt by any task to open (mount) a logical drive which corresponds to an IDE drive. During its initialization, the IDE Driver installs an Interrupt Service Procedure for the IDE controller interrupt. The driver creates an AMX semaphore which it uses to signal completion or termination of data transfers. Your AMX System Configuration Module must provide enough semaphores to permit one to be used by the IDE Driver.

Once the IDE Driver is able to access the IDE drives, it reads the partition information from each of the fixed disks attached to the IDE controller to construct its private representation of the logical drives available on the fixed disks. The IDE Driver ignores all logical drives beyond those specified in your IDE driver definition in your User Parameter File.

The IDE Driver is ready for use with AMX 86 in 80x86 environments with a PC compatible IDE controller. When used on non-PC platforms, the IDE Driver will require porting as described in this chapter.

Source code for the IDE Driver is provided separate from the AMX/FS File System in the following files.

FJnnnIDE.C	IDE Driver AMX/FS interface and control
FJnnnIDA.ASM	Data transfer support module
FJnnnIDB.C	AMX kernel interface

In addition to these modules, several board support procedures are provided by the AMX/FS File System in file *FJnnnBRD.ASM*.

The IDE Driver C source files and the AMX/FS board support module must be compiled and assembled to produce object modules *FJnnnIDE.OBJ*, *FJnnnIDA.OBJ*, *FJnnnIDB.OBJ* and *FJnnnBRD.OBJ*. These object modules must then be linked with your AMX application. This procedure is described in the AMX/FS Tool Guide.

Formatting IDE Drives

Each fixed disk drive attached to the IDE controller must be low level formatted and partitioned into one or more logical drives before the disk can be accessed by the AMX/FS IDE Driver. The IDE Driver does not provide this capability.

AMX/FS service procedure *fjmkfs()* can be used to transfer an MS-DOS file system to any of the logical drives on the fixed disks attached to the IDE controller.

Porting Issues - General

When used on non-PC platforms, the IDE Driver will require porting. Porting the driver for use with any IDE disk controller will be simple. However, to port the driver for use with a non-IDE controller will require significant change.

To port the IDE Driver for use in non-PC environments, you must edit the IDE Driver files to meet your IDE controller interface requirements. All sequences within files which require editing are marked as follows.

```
/* ===== Begin Modification ======= */
```

In file *FJnnnIDE.C*, adjust the definitions of the various timeout constants to match your drive specifications. By default the data transfer completion timeout is set to 12 seconds. Other time constants are IDE controller and drive specific.

In file *FJnnnIDB.C*, modify the definitions of the IDE controller I/O port number or device address. You may also have to modify the definition of the input and output macros which are used to transfer 8-bit values to and from external hardware devices.

Porting Issues - Interrupts

The AMX/FS IDE Driver includes an Interrupt Service Procedure ready for use with the particular version of AMX which you are using.

The AMX/FS 86 IDE Driver builds its own ISP root and installs the pointer to that root into the processor's Interrupt Vector Table.

The ISP Root provides access to the IDE Driver's Interrupt Service Procedure *fj_ide_isr()* in module *FJnnnIDB.C*.

In file *FJnnnIDB.C*, you must modify the definition of the IDE controller's interrupt vector number. If necessary, modify the IDE Driver support procedures in modules *FJnnnBRD.ASM* and *FJnnnIDA.ASM*.

5. AMX/FS Sample Program

5.1 Construction

The AMX/FS File System is delivered with a Sample Program which can be used on your target hardware to verify the proper operation of the file system. The Sample Program is an AMX application which, by default, uses the AMX/FS RAM Disk to exercise the AMX/FS File System. By simple edits to the Sample Program's User Parameter File, the Sample Program can be modified to reference a logical drive supported by any of the AMX/FS PC BIOS, UDD, floppy or IDE Drivers.

Source code for the Sample Program is provided with the AMX/FS File System in the following files.

`FJSAMPLE.C`	Sample Program
`FJSAMCON.C`	Console output
`FJ838BRD.ASM`	AMX/FS board support module
`FJSAMSCF.UP`	Sample Program User Parameter File
`FJSAMPLE.LNK`	Link Specification File
`FJSAMPLE.LOC`	Locate Specification File (if needed)

The Sample Program System Configuration Module `FJSAMSCF.C` must be generated from the User Parameter File `FJSAMSCF.UP`. This file is generated using the AMX 86 Configuration Generator as described in the AMX 86 Tool Guide.

The Sample Program's File System Configuration Module `FJSAMFCF.C` must be generated from the User Parameter File `FJSAMSCF.UP`. This file is generated using the AMX 86 Configuration Generator as described in the AMX/FS Tool Guide.

The Sample Program C source files, the configuration modules and the AMX/FS board support module must be compiled and assembled to produce the following object modules.

`FJSAMPLE.OBJ`	Main program
`FJSAMCON.OBJ`	Primitive console output
`FJ838BRD.OBJ`	AMX/FS board support module
`FJSAMSCF.OBJ`	Sample Program System Configuration Module
`FJSAMFCF.OBJ`	Sample Program FS Configuration Module

These object modules must then be linked and located using the Link and Locate Specification Files. The process, described in the AMX Tool Guide, is the same as that used to create any AMX application.

If you choose to use the AMX/FS PC BIOS, UDD, floppy or IDE Drivers, you will have to compile their modules as well. Edit the User Parameter File (see Chapter 5.3) and use the revised file to create an alternate FS Configuration Module. You will also have to edit the Link Specification File to link the required driver modules with the Sample Program.

5.2 Operation

The AMX/FS Sample Program consists of a `main()` program, a Restart Procedure `app_restart()`, an Exit Procedure `app_shutdown()` and a single task `maintask()`. The Sample Program provides an example of the correct calling sequence for most of the AMX/FS service procedures. The tests performed are designed to illustrate many of the features of the AMX/FS File System, albeit only using a single drive.

The Restart Procedure triggers the main task. The task initializes the RAM Disk and then performs its test sequence using the first available logical drive that it can find. This lookup process illustrates how the drive variables specified in the Logical Drive Table in the User Parameter File can be used to advantage.

The task creates a single directory and, within it, creates a series of nested subdirectories. Then, in the deepest nested directory, it creates a set of files and, using a variety of file handles, opens the files, writes to them and verifies that the writes have succeeded.

The task also verifies that the file pointer and file attribute manipulation features of AMX/FS are functioning properly. Finally, the task tests that all files and directories can be successfully renamed and deleted.

The test sequence is then repeated several times before AMX is called to shut down the Sample Program. AMX calls the application Exit Procedure which, in this simple example, does nothing. AMX then shuts down and returns to the `main()` function from which it was launched.

A number of parameters in the Sample Program module `FJSAMPLE.C` define the operational characteristics of the test. These symbolic parameters define the number of test iterations, the number of directories tested and the maximum subdirectory nesting depth within each of those directories. File size is also defined permitting simple adaptation to systems in which only a small RAM Disk can be provided.

RAM Disk

By default, a 96 Kb RAM Disk is preconfigured in the User Parameter File. However, by setting parameter `RDSPAGE` to `0` in the RAM Disk driver definition in the User Parameter File and altering the definition of symbol `RDLOCAL` to `1` in module `FJSAMPLE.C`, you can readily experiment with the dynamic creation of a RAM Disk.

Console Output

The Sample Program includes a primitive console output capability in file `FJSAMCON.C`. For AMX/FS 86, this console driver writes directly to the video memory of the PC.

If you cannot use the console driver as provided or cannot easily adapt it to your hardware configuration, you can alter the definition of symbol `K_VERBOSE` to `0` in module `FJSAMPLE.C`, thereby disabling any console output by the AMX/FS Sample Program.

5.3 Examples

The default AMX/FS Sample Program includes the file system parameters illustrated in Example 1 in its User Parameter File to allow it to operate using only a RAM Disk.

The remaining examples in this chapter show how simple it is to modify the User Parameter File to allow the sample program to use any of the other drivers provided with AMX/FS or available as options. Note that the RAM Disk remains present but is defined in the Logical Drive Table to be unused. Also note that the *DNFAT* parameter for unused drives could be set to *0* to conserve memory space.

Example 1:   96 KB RAM Disk

```
;           AMX/FS configuration and driver definitions
;
...FSYS            1,20,30,1
...FSFLP
...FSIDE
...FSPCB
...FSUDD
...FSRAM           4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV           ~0,sam_floppy,9
...FSDRV           ~1,sam_spare,9
...FSDRV           ~2,sam_ide,64
...FSDRV           ~3,sam_udd,64
...FSDRV           4,sam_ram,9
```

Example 2:   1.44 Mb Floppy Drive A

```
;           AMX/FS configuration and driver definitions
;
...FSYS            1,20,30,1
...FSFLP           1440,NONE
...FSIDE
...FSPCB
...FSUDD
...FSRAM           4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV           0,sam_floppy,9
...FSDRV           ~1,sam_spare,9
...FSDRV           ~2,sam_ide,64
...FSDRV           ~3,sam_udd,64
...FSDRV           ~4,sam_ram,9
```

Example 3: IDE Drive C

```
;           AMX/FS configuration and driver definitions
;
...FSYS             1,20,30,1
...FSFLP
...FSIDE            1
...FSPCB
...FSUDD
...FSRAM            4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV            ~0,sam_floppy,9
...FSDRV            ~1,sam_spare,9
...FSDRV            2,sam_ide,64
...FSDRV            ~3,sam_udd,64
...FSDRV            ~4,sam_ram,9
```

Example 4: PC BIOS Floppy Drive B (AMX 86 only)

```
;           AMX/FS configuration and driver definitions
;
...FSYS             1,20,30,1
...FSFLP
...FSIDE
...FSPCB            2
...FSUDD
...FSRAM            4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV            ~0,sam_spare,9
...FSDRV            1,sam_floppy,9
...FSDRV            ~2,sam_ide,64
...FSDRV            ~3,sam_udd,64
...FSDRV            ~4,sam_ram,9
```

Example 5: UDD Drive D

```
;           AMX/FS configuration and driver definitions
;
...FSYS              1,20,30,1
...FSFLP
...FSIDE
...FSPCB
...FSUDD             4,0,1
...FSRAM             4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV             ~0,sam_floppy,9
...FSDRV             ~1,sam_spare,9
...FSDRV             ~2,sam_ide,64
...FSDRV             3,sam_udd,64
...FSDRV             ~4,sam_ram,9
```

Example 6: UDD Drive A

```
;           AMX/FS configuration and driver definitions
;
...FSYS              1,20,30,1
...FSFLP
...FSIDE
...FSPCB
...FSUDD             1,1,1
...FSRAM             4,6,32
;
;           AMX/FS Logical Drive Table
;
...FSDRV             0,sam_floppy,9
...FSDRV             ~1,sam_spare,9
...FSDRV             ~2,sam_ide,64
...FSDRV             ~3,sam_udd,64
...FSDRV             ~4,sam_ram,9
```

This page left blank intentionally.

A. Typical Drive Specifications

The AMX/FS File System allows you to transfer an MS-DOS file system to a logical drive using AMX/FS service procedure *fjmkfs()*. This procedure receives as input a pointer to a drive format specification structure *fjxformat* which is defined in AMX/FS header file *FJ838KF.H*.

Typical drive specification parameters which are used to format floppy diskettes are shown in Figure A-1. Also shown are examples of the drive specifications for four fixed disk logical partitions. These examples are for guidance only and should not be interpreted as absolute specifications. You must use parameters which match your particular floppy diskette media or reflect the manner in which your hard disk has been partitioned.

Several of the fields in structure *fjxformat* require explanation. Field *xftsecpalloc* defines the granularity with which sectors on the drive are allocated to files. This parameter is often referred to as the cluster size.

Field *xftnumroot* determines the number of files and directories which can stem from the root directory.

The drive number field *xftdriveno* identifies which of the physical drives connected to the disk controller contains the logical drive being formatted. For all AMX/FS drivers, this value will be *0* or *1* since only two physical drives per controller are supported.

	Floppy diskettes				Fixed disks			
	360 Kb	720 Kb	1.2 Mb	1.44 Mb	20 Mb	80 Mb	512 Mb	2 Gb
xftdriveno	n	n	n	n	n	n		
xftmedia	0xFD	0xF9	0xF9	0xF0	0xF8	0xF8	0xF8	0xF8
xftsecpalloc	2	2	1	1	4	8	16	64
xftnumfats	2	2	2	2	2	2	2	2
xftsecrsv	1	1	1	1	1	1	1	1
xftsecpfat	2	3	7	9	44	88	255	255
xftnumroot	112	112	224	224	512	512	512	512
xftsecptrk	9	9	15	18	17	17	63	63
xftnumhead	2	2	2	2	4	8	16	16
xftnumcyl	40	80	80	80	612	1224	1024	4095

Figure A-1 Typical Drive Specifications

This page left blank intentionally.

AMX/FS 86 File System
User's Guide Index