# AIPS++ Portability

21 August 2003

**B.E. Glendenning**

Last modified: $Date: 2003/07/16 03:48:55 $ By: $Author: aips2adm $

# 1 Introduction

Portability is an important issue for AIPS++. If AIPS++ is insufficiently portable it risks dying an early death due to being tied to obsolescent hardware or software. On the other hand, if it takes an approach that is maximally portable, AIPS++ might become difficult to program in, or it might make such "lowest common denominator" assumptions about its host that it might be unable to take advantage of facilities that host might offer.

Comparison with the existing AIPS system might be instructive. First, it has to be noted that AIPS is a very portable system[1] and its portability has stood it in good stead. AIPS has survived three fundamental shifts in scientific computing environments that have killed off many other packages:

1. The shift to VMS from older mainframes (Modcomp). Computation and imaging were handled with quasi-independent hardware — array processors and "TV" display devices.
2. The shift to Unix mini-supercomputers. The computation moved into the host computer's vector units, or into parallel CPUs, while display continued to be on stand-alone display devices.
3. The shift to unix workstations. Computation moved directly into the host CPU, and display moved onto the workstation in its native windowing system. Networked computing — remote tape drives and displays — are also now supported at some level (but not distributed computation).

The problems which are often ascribed to problems caused by AIPS portability are:

1. The existence of an AIPS "operating system" which nearly precludes the user from using the host operating system commands on her data (e.g., "ls", "mv", *etc.*). Similarly, the system manager complains about having to maintain an AIPS operating system, when the native one is more powerful and flexible (for example, native *vs.* AIPS batch queues).
2. Portable FORTRAN is an unpleasant programming language that programmers were loathe to use.
3. The AIPS memory model is unduly restrictive.

In answer to the first point, it first should be pointed out that many users consider this to be a "feature" rather than a "bug"; users who run AIPS on disparate machines appreciate the uniform environment. Nevertheless, it is true that users should be able to use host tools (editors, OS commands, *etc.*) and generally "see" more of the host operating system and underlying filesystems than AIPS allows. The extreme case of this is that AIPS++ has a requirement that one user interface be the host operating system shell (C-shell, Korn Shell, ...).

The second point is more a reflection of FORTRAN than of AIPS (although arguments might be made that F77 could have been used earlier). In any event, it is certainly the case that C++ offers the programmer an astonishing amount of flexibility[2]. Until standard C++

[1] I once ran the AIPS Fortran source through the "f2c" FORTRAN to C converter, and it ran almost immediately. AIPS can typically be brought up on a new type of workstation with only one or two days of initial effort.

[2] In some circles, this might be described as "enough rope to hang herself with."

compilers are available (and remember — there is not yet a C++ standard) we may need to use a preprocessor to emulate features that not all compilers have.

In answer to the third point, the AIPS memory model was not unduly restrictive — in principle it could allow for dynamic memory allocation (and did for a time, on Cray machines). The real problem with AIPS is that the computational "layer" was too low; it doesn't easily encapsulate computation for arrays of higher dimensionality than one (vectors). It is clear that multidimensional arithmetic must be encapsulated (*e.g.,* multidimensional gridding). Since parallel machines are clearly going to be the next "wave" of computing, we must also port to parallel machines early in the AIPS++ life-cycle.

Portability is important. Successful portability is simply a matter of choosing interface layers that hide the details of a particular computational environment (this is a similar notion to "data hiding" in programming languages). If we are lucky, very thin layers on top of unix may suffice for the lifetime of AIPS++; if we are unlucky those layers will allow us to evolve into new environments. Note that although the strategy is the same as in AIPS, the level of the layers can be much higher as they can be built on top of underlying facilities like hierarchical filesystems, dynamic memory, and interprocess communications, which were not generally available when AIPS was first designed and written.

It should be noted that whilst several operating systems are being touted as possible successors to unix (*e.g.,* Windows NT), these are likely to possess system interfaces which conform to standards derived from unix, and the "thin layer" approach described above should be applicable, allowing small differences to be accommodated.

# 2 Operating System

## 2.1 Services

As some anonymous wag said, "the good thing about standards is that there are so many of them you can pick the ones you like." Since we are initially, and maybe for the entire life of AIPS++, interested in "Unix" operating systems I will only discuss standards developed in conjunction with Unix (although they are not necessarily tied to Unix). It must first be pointed out that "Unix" is a family of related operating systems with many different flavors: SVR4, OSF, Mach, BSD, *etc.* The various standards have been written to allow you know what functionality every conforming OS contains. The most commonly mentioned standards are POSIX.1[1], FIPS[2] , and X/Open[3]. The most important of these currently is POSIX.1, and I will concentrate on it here. There is, however, currently a lot of speculation in the trade press that the next X/Open standard might be very important.

In the following sections, I list some operating system features that will be clearly required by AIPS++, as well as some notes on implementing them on top of POSIX. An excellent book covering the topic of portable programming in Unix environments is *Advanced Programming in the UNIX Environment* by W. Richard Stevens (Addison-Wesley), 1992. Much of the following is drawn from this source as well as the POSIX.1 standard.

My goal has been to have layers which closely follow existing Unix practice, but would not be impossible to implement on other operating systems (albeit with some effort).

### 2.1.1 Files and Filesystems

We will require a layer which presents an interface to disk filesystems. Unix style special devices might appear in a similar layer, or be accessed via interprocess communications and server programs. It should be emphasized that these layers may still be below those that applications programmers normally use.

This file layer should have at least the following features:

`Hierarchical directory structure`
: The file layer should have a hierarchical layout with named files. POSIX.1 minimum limits are 14 byte filenames and 255 byte pathnames. Both of these are unfortunately small, and we should seriously consider having the file layer extend these with mappings for OS's which have these strict limits.

`Protection`
: Owner, group, and other permissions as in POSIX.1 should be available.

---

[1] *Portable Operating System Interface for Computer Environments*, developed by the IEEE. POSIX.1 is the operating system interface standard.

[2] *Federal Information Processing Standard*, produced by the US government. It is POSIX with some optional features made mandatory.

[3] X/Open is a group of computer vendors. The current (3rd) edition of their *Portability Guide* contains some extensions to POSIX.1.

`Directory operations`
It will be possible to list, search, and delete entries from directories. Information that POSIX.1 has associated with the file will also be available in the directory, such as file modification time, protection, and size in bytes.

`Links` Both symbolic and hard links should be available. Soft links have been added to POSIX.1 in an update; hard links have always been available.

`File I/O` Disk files will behave as if they are arbitrarily seekable, and that the file may be written at arbitrary positions, and that the file may be extended. These capabilities exist in POSIX.1.

`Asynchronous File I/O`
POSIX.1 does not currently have asynchronous I/O. Nevertheless, this is an important tool (and is available under all popular versions of Unix) so the layer should support it, even if it is a stub that blocks and always returns "READY" on some systems.

### 2.1.2 Asynchronous Processes

Generally, we will provide a layer where asynchronous processes may be created, signalled, listed, and destroyed. I don't believe that "fork" needs to be made visible to the application programmer, she merely needs to be able to start up independent processes. However, if this is incorrect, a fork binding can be provided.

A process has an environment which it passes to its children. A process has real and effective user and group permissions. A process with appropriate permissions may asynchronously signal or kill processes on the same host. Every running process on a given host has a unique identifier (which may be reused by a later process after the given process has finished).

All of the above is provided by POSIX.1.

### 2.1.3 I/O, IPC, and Networking

POSIX.1 has essentially nothing to say about networking and IPC, other than allowing for Unix-style pipes and FIFOs. AIPS++ will require much richer facilities to work in a networked world.

I assume that the lowest level IPC layer will work with raw bytes, and conversion to and from canonical data formats will take place at a higher level.

There are many different types of IPC available (shared memory, sockets, streams, *etc.*) on which the AIPS++ IPC layer may be built. Sockets are the most probable first implementation since they are generally available. Whatever the underlying implementation, the IPC layer should provide a logical view which provides for the following:

`Addressing`
You should be able to communicate either with a known process or with a named service (port) on any machine that is network accessible.

`Reliable communications`
The application programmer should not have to worry about lost or missing packets unless the communication link is broken.

`Bidirectional links`
: The IPC will appear to be capable of bidirectional communications, even if the implementation requires a pair of unidirectional links.

`Out-of-band data`
: The IPC layer will be capable of handling OOB data and of performing some signal when such has been sent.

`Selection of active links`
: Some facility will be available to alert the applications programmer as to which IPC links have activity upon them (like the BSD `select` system call).

## 2.2 Shells and Utilities

POSIX.2 is a standard which defines what shells and utilities must exist on conforming systems. These shells and utilities will generally be for system maintenance, source-code compilation, and program startup (*e.g.*, tape daemons) rather than direct implementation of AIPS++ functionality. While portability here is desirable, it is less critical than for the actual source code implementation.[4]

One problem with POSIX.2 utilities is that they are a "lowest common denominator" and may not offer adequate functionality. For example, the POSIX.2 `make` program is inadequate for complex multi-directory source code compilation (instead we use the `make` from the GNU project).

The approach we will generally take will be:

1. Where adequate, use POSIX.2 functionality.
2. Otherwise, attempt to use tools which are very portable (written to POSIX.1!).

In passing, I will note that there is a very general tool, `perl`, which is both extremely powerful for the types of operations covered in POSIX.2, and which is itself very portable (including VMS and MS-DOS). We may well evolve away from shell scripts to `perl` scripts at some point in the future.

[4] While AIPS itself is extremely portable, its associated compilation and maintenance tools are not all portable between VMS and Unix (but are in general between different flavors of Unix).

# 3 Environment

## 3.1 Native Word Size and Order

AIPS++ will have to live in a world where word order (big-endian *vs.* little-endian) and word format (two's complement integers, IEEE floating point) may vary between machines on the network.

AIPS is unable to share data on disk between hosts with different word orders since the data is read and written to disk in "local" order, and no central place knows all the file formats. (AIPS could be retrofitted to address this; however, it would be a large job). AIPS++ should get this right from the start. The alternatives are to either:

1. Read and write the data in a canonical format; or
2. write the data in the local format and be able to read all formats.

The first alternative is the cleaner approach; the cost is "on-the-fly" conversions for non-canonical systems. The experience of the BIMA Werong system is that these conversions have only about a 10% effect (and, of course, they will not normally be needed at all).

Sun has created a set of interfaces (XDR — *eXternal Data Representation*) for transferring data between a canonical format (IEEE big-endian) and "native" formats. These routines should be available on all machines of interest to AIPS++ (and the interface would be straightforward to replicate in any event). XDR should suffice for the time being, although we may ultimately need to replace it since it appears to do some things inefficiently (like only converting a single value at a time).

## 3.2 Compilers

In choosing C++ we have chosen a language without a standard[1]. While the C++ standard is coming, it is probably at least a year away. (C++ is being standardized by ANSI for ISO; so both standards will appear at the same time).

So far we have been using a preprocessor (the Texas Instruments "COOL" preprocessor) to allow us to use features like templates and run-time type identification consistently on the compilers currently in use in the AIPS++ project[2]. Within the next few months we hope to be able to use "native" templates. It is possible that we will need a preprocessor for some time, however, to mask the disparities in the abilities of the base compilers.

One critical requirement, *viz.,* language support for exception handling, is likely to take longer. For this reason, an exception handling system, similar in functionality to that proposed for the standard, has been implemented by Darrel Schiebel. It should be possible to migrate from this to the real system when it appears with fairly minimal effort.

---

[1] Had we chosen FORTRAN 90, we might have had a standard without many compilers!

[2] Which are: Sun CC and ObjectCenter (both CFront based compilers), GNU g++ version 2, IBM's XLC compiler

## 3.3 Windowing Systems and Graphics

Unfortunately the situation here is a bit confused. Even assuming that we are only interested in graphics[3] on workstations or X-terminals rather than on standalone devices, the strategy isn't entirely clear.

There is one policy question which must be answered before anything can be decided in this area: How much functionality should be in AIPS++, and how much in third-party packages? In the following discussion I assume that the answer is: "Image processing at about the level of AIPS and some basic 3D applications, although exotic work will be done in specialized packages."

I also assume that we are willing to tie ourselves to X-windows only on machines upon which we want to do graphics. The cost of this assumption is that if we ever need non-X solutions we will need to dedicate a larger amount of work in retrofitting the code than to have written it portably in the first place.

There are two things to consider for a graphics system:

1. The windows, menus, widgets, cursor control, *etc.*, that are used for user interaction and multi-window display; and
2. the "canvas" into which the graphics are drawn.

I think there is a clear response for the first point: InterViews is a freely-available, powerful C++ based X-windows toolkit that has been ported to other windowing systems, and for which tools like GUI builders are available.

Unfortunately, pure X-windows is not really adequate for high-performance graphics and there is no other candidate that is a clear winner. Consequently we will need to write a graphics layer that in turn can be bound to raw X, PEX, OpenGL, or whatever other graphics library is available. In this model, X-windows handles all the windows, widgets, *etc.*, and some other graphics library (might) handle the rendering, image processing, *etc.*. Such a layer should be straightforward to implement (considerably easier than trying to put a layer on top of the "interactive" parts of X-windows).

[3] Display graphics — hardcopy will need to be handled much as it is now. It should, however, share as much of the programmer interface as much as possible.

# 4 Memory

The language C++ ensures that AIPS++ will have dynamic memory available, however the operating system provides it. However, a policy for memory usage in AIPS++ is still required.

In AIPS, the policy was to allow all machines to run all problems (however slowly). The essential policy decision that made this possible was that all problems could run in memory available on all computers (64k words minimum). For image-plane work, this is roughly equivalent to saying that you can keep some small number of image *rows* in memory.

The ability to define abstract data types ins C++ should allow us to follow a similar policy, but hide much of the complexity of memory management from the "casual" programmer. For example, most modern computers can store a modest number of *planes*[1] in memory, where "a modest number" is *currently* about two, but this will surely increase during the lifetime of AIPS++, and so the ultimate image class should be capable of buffering large images according to the amount of physical memory available, transparently to the application using the class.

An 8kx8k image would occupy 256 Megabytes of (virtual) memory, so if this recommendation is followed memory usage will at times be dramatically increased over AIPS. So, as a general rule applications that use up to about 0.5 Gigabytes of virtual memory *for the very largest problems* should be acceptable. Of course, applications should only ask for as much memory as they need; if they can work efficiently with a "scrolling window" they should use such. Note that this requirement implies that AIPS++ will only operate on 32-bit[2] or larger machines, since one can only allocate storage in C++ whose size can be held by an `int`.

If AIPS++ is successful, it will have "casual" programmers programming within it. These programmers may not want to deal with memory management (although C++ should allow this to be easier than with FORTRAN), and may want to keep all of their data in memory. This should not be disallowed by the AIPS++ libraries; however, applications should not be accepted as "core" tasks until they have been altered to not exceed the above limit.

[1] Perhaps when *cubes* are to be held in memory it will be time for the successor to AIPS++!

[2] Actually, 30 bits would suffice.