System management for aips++

Part 2: activation, generation, and verification

Mark Calabretta
aips++ programmer group
1992/Feb/14

1 Introduction

Part 2 in this series of discussion papers deals with the activation of **aips++** and associated server processes, the generation of the **aips++** system from its source code, and also several topics which can be grouped loosely under the term "verification". It has been distilled from material presented in the "aips2-tools" email exploder between July and December 1991.

The subject of "activation" particularly includes the mechanism used for defining the **aips++** environment and starting **aips++** and its server processes. This includes batch processing as well as interactive. Some aspects overlap with that of the **aips++** user interface.

System "generation" encompasses all that is required to generate the **aips++** libraries, executables, scripts, documentation and any other system files from the **aips++** source code. In particular, this includes installation of **aips++** at end-user sites.

"Verification" refers to anything related to testing the correctness of **aips++** . In particular, debugging **aips++** source code and verifying that it produces sensible answers. Also included would be a tool or tools which check that source code conforms to a basic style of formatting, and perhaps applies it. Although the subjects of assertion testing, pre- and post-conditions, class invariants, and class test suites could be considered a part of verification, they are not considered in this document, but instead left for the discussion on general coding standards.

2 System databases

The local characteristics of an **aips++** installation will be described in system databases. Note that the term *database* in this context simply refers to a file or set of files which store information, much like the `passwd`, `group`, `services`, etc. files in unix. The `SP` (system parameter) and `ID` (imaging device) files in $\mathcal{AIPS}$ are further examples of such databases. The `SP` file stored information on peripherals such as data disks (including disk bookings and `TIMDEST` limits), tape drives, TV, graphics, print and plot devices, the (pseudo-) array processor system, access restrictions, local system identification, and so on. The `ID` files stored a detailed description of each image display device, and were themselves used as a lock file for arbitrating TV access. **aips++** will probably not need to duplicate the $\mathcal{AIPS}$ parameter set, but some databases certainly will be needed.

A problem with the $\mathcal{AIPS}$ system databases came to light in the late 1980s when multiple-host installations of $\mathcal{AIPS}$ in a network environment started to become the norm. The design implicitly assumed a single-host installation, the norm in the preceeding decade. Having a separate `SP` file for each host did not cater for large areas of commonality between hosts at the same site. For example, changing a system parameter common to *all* machines at a particular site required changing the `SP` file for *each* individual machine. The fact that `SP` and `ID` files could only be modified by using special purpose programs was an added inconvenience.

These problems can be circumvented in **aips++** by using a heirarchical scheme based on `ASCII` text files. Each of a sequence of `ASCII` files would be read in a set order. The first file in the sequence would

contain a complete set of default definitions. Subsequent files would refer to an ever more restrictive domain of applicability and contain definitions overriding those of preceeding files. Given the directory heirarchy proposed in part 1 of this discussion document, the following sequence suggests itself

```
$AIPS/$ARCH/$VERS/aipsrc
$AIPS/$ARCH/$VERS/$SITE/aipsrc
$AIPS/$ARCH/$VERS/$SITE/$HOST/aipsrc
```

where the variables (not necessarily environment variables) have the following meanings

```
AIPS    ...root of the aips++ directory tree (e.g. /aips++)
ARCH    ...host architecture (e.g. sun4, ibm)
VERS    ...released (base) or development (apex) version
SITE    ...site name (e.g. aoc)
HOST    ...host name (e.g. baboon)
```

The first file in the sequence **$AIPS/$ARCH/$VERS/aipsrc**, containing default definitions, would be distributed with **aips++** and not be changed in the installed system. Instead, any site specific parameters would be defined in **$AIPS/$ARCH/$VERS/$SITE/aipsrc**. The hosts within each site would each have their own **aipsrc** file, and typically this would be used for attached peripherals. It has been argued that a **~/.aipsrc** file should also be provided for users to override system defaults. This would require a mechanism to prevent them from circumventing access restrictions. The routine which reads the databases should allow that the host, and possibly the site **aipsrc** files be empty or missing.

The fact that site-specific databases reside in a subdirectory with a site-specific name allows for easy maintenance of **aips++** among multiple sites belonging to a single institution. Given that a particular site holds the master copy for an institution, all that should be required to keep the slave copies up-to-date is to copy the directory tree from the master. The fact that the master and slave sites have different names will prevent the slave's databases from being overwritten.

3 Activation

It is clear from the user requirements that it must be possible to run **aips++** tasks from an ordinary user account rather than a generic "**aips**" account, and from a unix shell, such as Bourne shell or C shell. These requirements have a significant bearing on the way that task activation is implemented. In particular, it means that tasks must be able to determine system parameters for themselves at run-time. For example, a task might need to determine the amount of memory installed in the machine on which it is running (this would be stored in **$AIPS/$ARCH/$VERS/$SITE/$HOST/aipsrc**). It is unrealistic to suppose that all information of this type be translated into environment variables defined from the user's login script, so the **aipsrc** databases themselves would have to be consulted at run-time. That is, an **aips++** task or script which needs to know about devices etc. will read the **aipsrc** files as required.

The information stored in the **aipsrc** databases could be said to be static in that it does not change between invokations of **aips++** tasks. However, some parameters may change between a user's login sessions, in particular, **$ARCH** and **$HOST**. Moreover, **$PATH**, must be known in order to start a task, and **$AIPS**, **$VERS**, and **$SITE** are required to find the **aipsrc** files themselves. Information of this kind would be stored in "activation" databases.

It would be desirable to minimize the number of environment variables that need to be defined in order for a task to be activated. The most basic requirement is that the user's **PATH** be modified to include the **aips++** binaries and scripts. In general this would involve evaluating **$ARCH** and is more than can be done by a single line entry in a user's login file, unless it be an invokation of a special purpose script, i.e.

```
. /aips++/aipsinit.sh
```

for Bourne-like shells, such as *sh*, *bash*, *ksh*, and *zsh*, or

```
source /aips++/aipsinit.csh
```

for C shell-like shells, such as *csh* and *tcsh*. Although it is possible to determine a host's architecture by indirect means (as does the `~pvi/wave/bin/arch` script in *PVwave*), these are rather kludgy and a `$AIPS/hosts` database would be needed to resolve `$SITE` as well as `$ARCH`. `$VERS` should default to `base` unless overridden by a command-line argument to the `aipsinit.[c]sh` script.

Apart from redefining the user's path, the `aipsinit.[c]sh` script must also define `$AIPS` as an environment variable so that any **aips++** task can find the `$AIPS/hosts` database and rederive `$ARCH`, `$SITE` and `$HOST` for itself, since it will need these to find the `aipsrc` databases. Since the users specify `$VERS` at login time, it must also be recorded by `aipsinit.[c]sh` as an environment variable. Alternatively, it may be more efficient to store `$ARCH`, `$SITE` and `$HOST` as environment variables also, or instead have one single environment variable which stores all five components.

As discussed below, **aips++** will probably use server processes for image display, remote tape access, and other functions. It would be best to free **aips++** users from responsibility for the activation of these servers; tasks should start them automatically where required. The *inetd* mechanism, in which a server is activated when a client requests a connection, would be ideal for this purpose. However, it would require a root installation on all server machines and this is inadmissable. One solution would be for the client to try to `connect()` to the server and, if this fails, try once to `system()` an *rsh* script which activates it. Certain implementation problems have been glossed over here. Suffice it to say that efficient, automatic activation of servers, particularly image display servers, may be challenging.

The problem of assigning tcp or udp port numbers to network services has been discussed at length in the aips2-tools exploder. There is a potential for conflicts if port numbers are adopted arbitrarily, but registration of port numbers in the `/etc/services` file would require root privilege and so is inadmissable. **aips++** therefore needs a mechanism for the flexible allocation of port numbers, and to this end, a site-specific set of numbers for each service could be specified in the `aipsrc` databases. When a task invokes a server via *rsh* as described above, the port number would be passed to the server as a command-line option.

Where **aips++** tasks are being initiated from an interactive unix shell, users will be required to implement multi-tasking for themselves by backgrounding and foregrounding jobs in the usual way. Multi-tasking should probably be implemented automatically within the **aips++** shell itself.

One of the advantages of providing for **aips++** tasks to be run from a unix shell is that it provides a simple method for creating procedure scripts. Within these scripts it is conceivable that non-**aips++** tasks might be interspersed with those of **aips++** . These scripts would also be suitable for submission to the host operating system's batch facility. However, there was a clear requirement in the user specifications for a batch system in **aips++** , and given the poorness of some unix batch systems, **aips++** will probably have to implement its own batch system eventually.

Any procedure scripts required in **aips++** , for example those required for activating servers, may be written in *perl*. The simple `aipsinit.[c]sh` Bourne shell and csh scripts which define environment variables would still be required. Likewise the scripts used to define directory logicals for programmers who want them.

Given that **aips++** will be installed and provided at many sites by a centralized management, a standard mechanism for sending messages to users should be included in the distribution. A simple system called `GRUNT` has been operating for some time within $\mathcal{AIPS}$ at the ATNF and has proved invaluable. It records which users have read each message so as to avoid printing the same message twice. We have found that users are much more likely to pay attention to messages if the only messages they receive are new ones, although they still have the option of reviewing old messages via the "`NEWS`" command if they want. `GRUNT` allows for multiple message classes, and for deactivating old messages without deleting them so they can be kept as a record. It does, however, require that users run from their own account so they can be identified. We will probably also want to integrate it with the documentation system.

4 System generation

The discussion on system generation starts with the question of how imported `FORTRAN` libraries such as `PGPLOT`, `SLALIB`, etc. are to be treated. It was suggested in the aips2-tools exploder that these be con-

verted to C/C++ by using *f2c* followed by hand editing. However, several contributors felt uncomfortable with this, specifically because

- Conversion to C could introduce subtle bugs, particularly in numerical routines.
- FORTRAN is more easily vectorizable than C and number crunching packages converted to C might suffer.
- These libraries can be expected to evolve over time and the conversion would have to be done afresh for each release.
- Patches couldn't be applied mechanically.
- Hand-editing, if required, allows an added opportunity for bugs to creep in, especially since it probably wouldn't be done the same way for different releases.
- If a bug is discovered in a library routine it will take some time to ascertain whether it was native or introduced.
- It will consume somebody's time.

C++ does not have a calling convention for FORTRAN routines but it does provide for C. One alternative to conversion, therefore, would be to write a set of wrapper C-routines for each library which call the FORTRAN routines. However, in the absence of a universally accepted mechanism for calling FORTRAN from C, we would have to allow that these wrapper routines be operating system specific.

The programmer environment for **aips++** must support private workspaces and would be implemented via a set of environment variables. In particular, these must define the host architecture of the machine the programmer is currently working on, and transparently lock out the possibility of mistakenly interfering with the binaries for any other architecture. When a library or executable is recompiled it should only affect the `lib`, `bin` or `tmp` directories for the current architecture. Likewise, any code search paths should be resolved automatically for the current architecture without programmer intervention.

Logical names should be assigned to all **aips++** directories soley for the purpose of making it easier to move around the **aips++** directory tree. They should only be defined at the discression of the particular programmer, since many programmers already use shells or utilities (such as *filemgr* in SunOS) for moving around directory trees. In particular, the **aips++** build mechanism should not rely on these logicals being defined.

Compilation of all source files, including libraries, executables and online documentation should be under the control of the *make* utility. There are almost as many flavours of *make* as there are flavours of unix, some of them quite primitive. The approach adopted in POSIX.2 has been to define a very restricted set of capabilities which are basically a subset of all known *make*s. The *make*script itself must be POSIX.2 compliant and this is enforced by adding a `.POSIX` target. In particular, POSIX.2 *make* does not allow the `shell` function, source code search paths, or parallel execution options of GNU*make*. I believe that this will be too restrictive for **aips++** , and we should adopt GNU*make* in toto. This will still be within the bounds of POSIX compliance since GNU*make* itself is POSIX.1 and POSIX.2 compliant.

With Q- and Z-routines, and possibly wrapper routines for FORTRAN libraries, **aips++** will need a source code search path mechanism. GNU*make* provides this in the build phase, but we will need a facility for finding files independently of recompiling. This could well be implemented as a special target in the makefiles themselves.

The makefiles will need to be tailored for particular operating systems or sites, but we would really like to avoid having to edit makefiles to tailor them for a particular installation. One way of doing this may be to construct the makefiles in such a way that they look in site-specific database(s) for local macro definitions. If this proves unworkable, we may have to abandon *make* and revert to *imake*, the utility used for installing X-windows. However, writing *imake* configuration templates seems to be sufficiently complex that we will probably only want to take it on as a last resort.

Unlike the usual practice with makefiles, those for **aips++** must be careful to store object modules and other intermediate products of compilation in an architecture-specific area rather than in the same

directory as the code. The reason is simply that the code areas will be visible to machines of different architectures, two of which could conceivably be running *make* at the same time.

Another area where potential clashes can occur is when programmers on two different machines of the same architecture attempt to update an object library at the same time. The mechanism used in *AIPS* only accounted for two different programmers on the same machine, so something more sophisticated will be required. GNU*make* does not provide a solution to this problem, and unless someone can think of a better idea, we will probably have to use a variation on the *AIPS* scheme incorporating the machine name as well as the process id in the name of the lock file. Since *rsh* to a crashed host usually hangs, *ping* should be used first to check that it's still alive.

Since libraries are the only sensible way to manage link lists, object modules for static executables should be stored in archive libraries and the *.o files deleted. However, **aips++** should provide for sharable libraries [1] to be used optionally where operating systems provide this facility, as does SunOS. Normally speaking, sharable objects must be reconstructed from a complete set of their constituent object modules. Unless there was a requirement to link some executables statically, there would be no point in packaging and unpackaging the object modules in archive libraries whenever an object module changes, and they should therefore be kept in atomic form. This makes updating sharable objects very fast.

The strategy to be used in designing the makefiles must be considered carefully, otherwise we will find that recompiling a single application (to debug it for example) will take ages while all of the modules in all of the libraries are checked to see whether they're up-to-date. Dependencies for object libraries should follow a scheme where

- Each object library depends on all of its modules.
- Each module in an object library depends on all of its sources.

Concerning the second point, it may be simpler and more efficient in the long run for class implementations to depend on all class header files rather than attempt to list dependencies explicitly. Dependencies for applications should be as follows

- Each application should depend on all of its sources.
- Each application should depend on the last modification time of its libraries, but not so as to cause them to be remade.

Concerning the last point, timestamp files could be used if necessary to record the last update time for the libraries. In order to re*make* the whole of **aips++** , the libraries should be made, then the applications.

In principle **aips++** may have to be completely rebuilt each night, especially in the early stages when class header files might be expected to change on a daily basis. The rebuild would happen automatically at consortium sites after the latest revision of the code had been fetched, and will be required to have been completed by the following morning. Eventually this job may become too much for one machine, and we should investigate the possibility of spreading the load over several machines. GNU*make* allows parallel execution, but only within the same machine. However, this seems like a promising point to start from.

Concerning support for multiple devices, *AIPS* originally used alternate `LOAD` (`bin`) areas to support different types of device of a particular kind within a single system, principally TVs and array processors. Later, with the advent of networks, a much more flexible model was adopted - that of a device server. This model was developed for image display devices with `TVMON`, then `SSS`, and later `XVSS` and `XAS`. A tape server, `TPMON` has recently been added, and **aips++** can expect to have compute servers as well. The aips2-tools exploder reached a consensus on avoiding alternate load areas. The device server model

[1] The use of sharable libraries constrains the way memory is used within object modules. In particular, uninitialized data should not be statically declared (i.e. as large arrays) but rather allocated from the heap so that the uninitialized data segment for the sharable object as a whole is kept as small as possible. Otherwise, large amounts of static memory may needlessly be allocated to processes which do not require it. However, this topic is properly the domain of coding standards and is not addressed here.

is the obvious solution, and has the obvious benefit of providing net access. The use of shared memory may increase the efficiency of servers running on the same machine as their clients.

Support for multiple plot devices is a somewhat different problem in that the devices are only *logically* different, i.e. the output is always ASCII text, only the graphics commands are different. PGPLOT handles multiple devices by using run-time switches for different device libraries. As delivered, one decides at install-time whether or not to include support for a particular device in PGPLOT by hand-modifying the GREXEC routine. Rather than modify code, however, **aips++** should handle this at the makefile level by conditional compilation, e.g. (using *cpp* for FORTRAN!)

```
#ifdef GFDRIV
1     CALL GRDRIV (...)
      RETURN
#else
1     CALL ERROR ('No GFDRIV support')
      RETURN
#endif

#ifdef IMDRIV
2     CALL IMDRIV (...)
      RETURN
#else
2     CALL ERROR ('No IMDRIV support')
      RETURN
#endif

    :
    :
```

The key to the PGPLOT scheme is that only a very limited number of routines (only one in PGPLOT) should contain the device switch. One might remove support for devices which are not available simply in order to reduce the size of the executables. In applying this mechanism in the general case, however, the particular device libraries simply may not be available, and this would have been the situation with TVs and APs in *AIPS*.

Documentation, whether it be online help, printed manuals for users, or class documentation for programmers, will be discussed elsewhere. However, the documentation system will be more sophisticated than in *AIPS*, and in general will entail compilation from its sources. For example, it is intended that **aips++** applications will define their input parameters within their prologue section, and a special purpose tool will be required to extract the definitions and add them to the documentation for the task. The documentation may need to be compiled in different ways to serve as input for different purposes, such as an X-windows client, a terminal, or for printing. Compilation of the documentation from its sources should be under the control of makefiles. A separate copy of the compiled documentation should be maintained for each machine architecture to ensure that the $AIPS/$ARCH/... subtrees contain an independent and fully functioning system as described in part 1 of this document.

5 Installation

Many aspects of the installation process have been discussed in passing in the preceding sections. However, this topic is of sufficient importance that it deserves separate attention.

In the first instance an end user site will obtain a copy of **aips++** from their local **aips++** consortium member, via anonymous *ftp* or tape. Many sites will be able to copy the **aips++** binaries directly, thereby saving them the trouble of recompiling the system. These binaries must therefore be "exportable" in the sense that they contain no hard-coded site-specific information. The installation would thus be completed by encoding the site-specific information into ASCII databases such as the $AIPS/hosts and $AIPS/$ARCH/$VERS/$SITE/aipsrc files.

Inevitably, many sites will not be able to take advantage of the binary distribution, either because they have incompatible hardware, or incompatible releases of their operating system or utilities. These sites will have to rebuild **aips++** from its source code, and we must ensure that the **aips++** system is completely constructible from the **$AIPS/$VERS/code** subtree. This includes architecture-specific procedure scripts (if any). Apart from defining site-specific parameters in databases like those mentioned above, the installation would first involve editing a file used by the **aips++** makefiles, **$AIPS/$ARCH/$VERS/$SITE/sysconf**, containing a more detailed description of the system. This might include the location of various public domain libraries or utilities required by **aips++** and which the site had already installed, but in a place different to where the makefiles expect to find them by default. Template versions of this **ASCII** descriptor file should contain defaults which the installer may change as required by using a text editor. After editing the **sysconf** file, a single recursive *make* should rebuild the whole of **aips++** .

6 Verification

There was a protracted discussion concerning debugging strategies in the aips2-tools exploder. I do not intend to retrace all of the ground covered there, but simply propose a solution to the problems raised, the main one of which is that debugging optimized code is difficult for most existing debuggers, and impossible for some, for example Sun's. The only debugger which is claimed to work well at even the lowest level of optimization is Convex's CXdb, and even it seems less than perfect. However, given that optimizers for RISC machines can achieve a speedup of 100% or more, **aips++** simply can't afford to do without it, Q-routine encapsulation not withstanding. An added complication is that the optimization may also introduce compiler-based bugs. The solution proposed here has been implemented in *AIPS* at the ATNF and appears to be quite workable. It is assumed that the working enviroment is one where **aips++** is used simultaneously for production and development.

The idea is simply to have two sets of object libraries, one compiled optimized/no-debug, the other unoptimized/debug. The executables would normally be stored in optimized/no-debug form, but when debugging is required, it would be a simple matter to rebuild an executable in debug mode, since all libraries are present. Recompilation of static executables before debugging is good practice since it removes the possibility of an executable having been linked with a stale library. Debug executables would not normally be left in the system, and as set up at the ATNF, they are automatically deleted after three days.

If a compiler optimization bug is suspected, it should be a simple matter to do an unoptimized recompilation, rerun the task, and compare the output. If compiler optimization bugs are found, the offending functions may be added to a list of routines not to optimize.

Sharable libraries, if available, should only be used for the optimized executables. The debug libraries should be static for the following reasons

- It avoids confusion. If several programmers were debugging a number of applications or functions independently, the last thing they'd want is to have sharable objects changing while they worked.
- Since only a few executables should exist in debug form at any time, the use of sharable libraries is actually *less* efficient in terms of disk usage.

In other words, the two main features which make the use of sharable libraries attractive for the production system make them undesirable in the development system.

Some participants in the aips2-tools exploder felt it was important that it be possible to have a debugger start automatically from an exception handling routine. This can be done for the VMS debugger, but not so for any unix debuggers. Also, it may not be appropriate for users running an **aips++** task to be thrown into a debugger automatically. A simple alternative would be for the exception handler simply to ask a question, thereby allowing time to start up a debugger and connect to the process. Something like

```
"Exception xxx occurred, debugger may connect to
 process yyy now, otherwise hit <cr> to continue:"
```

Naturally this should only happen in interactive mode!

aips++ will have a set of coding standards which can be considered to be of two types, those of style and those of substance. The latter, which might include the use of C++ constructs that a putative *lint++* would deal with, are beyond the scope of this discussion. The former, category includes what are sometimes called "typing" standards and deal with indentation, the placement of braces, and so on. Although it is also outside the scope of this document to define these standards, it is relevant to mention here the need for a tool which ensures that code conforms to, and if possible applies these standards. For example, a version of the *indent* utility exists which will format C++ code, and the **aips++** typing standards could well be defined as a standard set of options to this utility.

A related topic is that of pretty-printing C and C++ source code. The *tgrind* utility is built on TEX and operates on plain source code. It uses different fonts to distinguish between comments, keywords, strings, and the rest. Although fairly limited in scope, *tgrind* does make source code more aesthetically pleasing and arguably easier to follow. Although it uses an unsuitable proportionally spaced TEX font by default, it can be modified to use the fixed-spaced Courier POSTSCRIPT font.