AIPS++ System Manual
For users, programmers and administrators

Mark Calabretta

1997/06/05
2000/09/22

The AIPS++ consortium may be reached by email at aips2-request@nrao.edu. The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd., Charlottesville, Va. 22903-2475 USA.

Contents

Chapter 1

Overview

This chapter [1] provides an overview of the AIPS++ system.

1.1 The AIPS++ directory hierarchy

In the following discussion of the AIPS++ directory hierarchy we will assume that AIPS++ has been installed in directory `/aips++`. This is the preferred location, but in practice the AIPS++ "root" directory can reside anywhere. The root directory is generally referred to as `$AIPSROOT`. Many other AIPS++ directories have standard variable names as listed in §1.2, p8.

The major subdirectories of `/aips++` are `code` (`$AIPSCODE`), `docs` (`$AIPSDOCS`), and one or more architecture-specific subdirectories with names such as `sun4sol_ntv` and `alpha_gnu` which contain the AIPS++ system - that is, everything needed to run AIPS++, including executables and sharable objects. These architecture-specific subdirectories are referred to collectively as `$AIPSARCH`.

1.1.1 Code directories

The directory hierarchy beneath `/aips++/code`, or `$AIPSCODE`, consists of a collection of "packages" which are contained in separate subdirectories. The principle package is `/aips++/code/aips` which contains the source code for a C++ class library required by all other packages. It also contains applications common to all areas of astronomical data processing.

The `dish`, `synthesis`, and `vlbi` packages contain standard classes and applications common to single dish, aperture synthesis, and VLBI data processing tasks from all radio telescopes. Apart from a dependence of the `vlbi` package on the `synthesis` package, inclusion of these standard packages in an end-user installation is optional.

A `contrib` package is the place for source code contributed from the AIPS++ user community for redistribution with AIPS++. If found to be generally useful, code from the `contrib` package may eventually be merged into one of the standard packages, but otherwise it is unsupported.

A `trial` package contains source code written by members of the AIPS++ consortium but not yet accepted for inclusion in the standard distribution.

[1] Last change: Id: Overview.tex,v 19.0 2003/07/16 04:18:27 aips2adm Exp

Each AIPS++ consortium member is also entitled to maintain a package for data processing applications specific to its telescope(s). The sources for these classes and applications reside in the consortium-specific packages: `atnf`, `bima`, `drao`, `nfra`, `nral`, `nrao`, and `tifr`. These may or may not use the standard `dish`, `synthesis`, and `vlbi` packages, and their installation is also optional.

The standard AIPS++ packages (`aips`, `dish`, `synthesis`, and `vlbi`) contain an `implement` subdirectory which contains class header and implementation files, a `fortran` subdirectory which contains FORTRAN subroutines, an `apps` directory containing applications, and `scripts` and `data` subdirectories for package-related procedure files and system data such as standard colourmaps, calibrator lists and source catalogues.

The `implement` and `fortran` subdirectories may contain module subdirectories which serve to collect software "modules" in one place. For example, the `Tables` module contains all class header and implementation files pertaining to the `Table` class, and classes derived from, and related to it. The `implement` directory and all module subdirectories may also contain a `test` subdirectory which contains one or more self-contained test programs specifically for the module.

All files associated with an AIPS++ application reside in a subdirectory of the `apps` directory of the same name as the application. Each application must reside in its own subdirectory.

The substructure of the consortium-specific packages is left entirely to AIPS++ consortium members to determine.

There are a number of other subdirectories of `/aips++/code` which are unrelated to packages. The `install` subdirectory contains all of the utilities required to install and maintain AIPS++ as discussed in this document. `doc` contains AIPS++ documentation sources, including the AIPS++ "specs", "memos", and "notes" series, and reference and design documentation in the corresponding subdirectories.

Also below `/aips++/code` is an `include` subdirectory which contains symbolic links to the `implement` subdirectories for each package. The purpose of these symlinks is to allow AIPS++ includes to be specified as "`#include <package/Header.h>`" by adding `-I/aips++/code/include` to the include path.

On AIPS++ consortium installations an additional `admin` subdirectory of `/aips++/code` contains files relating to the administration of the AIPS++ project.

The `/aips++/code` directory hierarchy appears as follows:

```
 :                                          +--- App1 ----
 :                           +--- apps ----+--- App2 ----
 |                           |              +--- .... ----
 |                           |
 |                           |              +-- Module1 --+--- test ----
 |                           +- implement -+-- Module2 --+--- test ----
 |                           |              +--- .... ----
 |                           |              +--- test ----
 |                           |
 |             +--- aips ----+-- fortran --+...
 |             |             +-- scripts --
 |             |             +--- data ----
 |             |
```

```
         |             +--- dish ----+...
         |             +- synthesis -+...
         |             +--- vlbi ----+...
         |             +-- contrib --+...
         |             +--- trial ---+...
         |             |
         |             +--- atnf ----+...
         |             +--- bima ----+...
         |             +--- drao ----+...
         |             +--- nfra ----+...
         |             +--- nral ----+...
         |             +--- nrao ----+...
         |             +--- tifr ----+...
/aips++ -+--- code ----+
         |             |             +- codedevl --
         |             |             +- codemgmt --
         |             +-- install --+- docutils --
         |             |             +- printer ---
         |             |             +-- <arch> ---
         |             |
         |             |             +-- design ---+...
         |             |             +--- html ----
         |             |             +--- memos ---
         |             |             +--- notes ---
         |             +---- doc ----+-- papers ---
         |             |             +-- project --
         |             |             +- reference -
         |             |             +--- specs ---
         |             |             +--- .... ----
         |             |
         |             +-- include --
         |             |
         |             |             +- personnel -
         :             +--- admin ---+- projects --
         :                           +-- system ---
```

1.1.2 Documentation directories

The `/aips++/docs` subdirectory, or `$AIPSDOCS` (§1.2, p8) contains ASCII, HTML and POSTSCRIPT documents compiled from the sources in `/aips++/code/doc` whose directory structure it shadows:

```
         :             +--- aips ----+...
         :             +--- dish ----+...
         |             +--- .... ----+...
         |             |
         |             +-- design ---+...
```

```
        |            +--- html ----
        |            +--- memos ---
/aips++ -+--- docs ----+--- notes ---
        |            +-- papers ---
        |            +-- project --
        |            +- reference -
        :            +--- specs ---
        :            +--- .... ----
```

`html` files compiled from inline comments in the C++ source code are deposited in the various package-specific subdirectories.

1.1.3 System directories

The AIPS++ system directory hierarchy is created in the first instance when AIPS++ is installed (see §2, p11), and maintained thereafter by the `sysdirs` target which is invoked by `allsys` in the top level `makefile` (p162).

Except for the `aipsinit` (p31) files in `$AIPSROOT`, the AIPS++ system is completely self-contained within the architecture-specific subdirectory, referred to as `$AIPSARCH` (§1.2, p8). In this context "architecture" should be interpreted to include variants in the operating system version and compiler.

In practical terms, the fact that the AIPS++ system does not rely on anything in the `$AIPSCODE` directories allows the source code to be deleted after the AIPS++ installation is complete in a production-line system.

At an AIPS++ development site with machines of several architectures where the source code must be retained, the strict separation of code from system provides for the `$AIPSARCH` tree to reside on a machine of the corresponding architecture without duplication of the `code` directories. With thoughtful unix filesystem management it also allows that the AIPS++ system for one architecture may remain available even if the server for any other architecture has crashed.

The system directories have the following structure:

```
        :            +---- lib ----+...
        :            +---- bin ----
        |            |
        |            +-- libdbg ---+...
        |            +-- bindbg ---
        |            |
        |            +-- bintest --
        |            |
        |            +-- libexec --
        |            |
        |            +---- aux ----
        |            +---- tmp ----+...
        |            |
        |            |             +--- info ----
/aips++ -+--- (arch1) --+---- doc ----+--- man1 ----
```

```
    |             |             +--- cat1 ----
    |             |             +---  :
    |             |
    |             |             +-- (host1) --
    |             +-- (site1) --+-- (host2) --
    |             |             +--   :
    |             |
    :             +-- (site2) --+...
    :             +---   :
```

The `lib` directory contains optimized static object libraries and possibly sharable objects. It sometimes also contains a subdirectory which serves as a C++ template repository. The `bin` directory contains AIPS++ system scripts and optimized applications. It is added to the `PATH` environment variable by the ***aipsinit*** (p31) scripts.

The `libdbg` and `bindbg` directories contain debug versions of the libraries and executables. The `bindbg` directory is not usually populated but serves as the temporary residence for executables which are in the process of being debugged.

The `bintest` directory is used temporarily to store test executables and test results, and `libexec` contains scripts of various kinds which are not meant to be executed directly but are instead included by other scripts.

Files which are produced as intermediaries of system generation are cached in the `aux` directory. In particular, it includes dependency lists generated by the makefiles. Temporary storage is provided during a rebuild beneath the `tmp` directory. The structure and usage of the `tmp` directory hierarchy is soley the concern of the AIPS++ makefiles. It contains subdirectories specific to each AIPS++ package (see §1.1.1, p1).

Online documentation is contained in the `doc` subdirectory. This includes unix manual pages and help files. The ***aipsinit*** (p31) scripts add this directory to the `MANPATH` environment variable if it is defined at the time that ***aipsinit*** is invoked.

Finally, the `$AIPSARCH` directory contains site subdirectories which contain site-specific `aipsrc` and `makedefs` files (see §1.3, p9), and possibly host-specific subdirectories which in turn contain host-specific `aipsrc` files. Multiple site-, and host-specific directories were provided to make it easier for a central site to administer AIPS++ for a collection of remote sites. If properly configured, it should allow a verbatim copy of the AIPS++ system at the central site to be downloaded at the remote site with only a minimum of reconfiguration required.

1.1.4 RCS directories

The AIPS++ RCS directories are maintained only at AIPS++ code development sites. They are not provided with the end-user AIPS++ distribution.

The one true AIPS++ master RCS repository resides on host *aips2.nrao.edu* at NRAO's Array Operations Center in Socorro, New Mexico, USA. This is the working copy for AIPS++ code management, consortium sites access this one way or another for checkin and checkout (see §9, p89).

The `master` RCS directory hierarchy is the same as the `code` hierarchy described above except that the master has an `etc` subdirectory which is used for central administrative purposes (see

§13, p263). In order to reduce the amount of network traffic involved in a checkin/out the master only contains revisions for the current and preceding major AIPS++ version numbers (see ***avers*** (p113)). A separate `archive` repository contains all revisions other than those for the current major version number.

```
         :             +--- etc -----
         :             |
         |             |             +--- apps ----+...
         |             |             +- implement -+...
         |             +--- aips ----+-- fortran --+...
         |             |             +-- scripts --
         |             |             +--- data ----
         +-- master ---+
         |             +--- dish ----+...
         |             +- synthesis -+...
         |             +--- vlbi ----+...
         |             +-- contrib --+...
         |             :
---------+
         |             +--- etc -----
         |             |
         |             |             +--- apps ----+...
         |             |             +- implement -+...
         |             +--- aips ----+-- fortran --+...
         |             |             +-- scripts --
         |             |             +--- data ----
         +-- archive --+
         |             +--- dish ----+...
         |             +- synthesis -+...
         |             +--- vlbi ----+...
         :             +-- contrib --+...
         :             :
```

Most AIPS++ consortium sites have the master RCS repository remotely NFS mounted beneath `$AIPSROOT`, and some also maintain a copy of the archive repository.

All AIPS++ consortium sites, including that at Socorro, maintain a slave RCS repository which is regularly updated by the AIPS++ code distribution system (§8, p73) so that it keeps track of the master (aside from the `master/etc` directory). This slave RCS repository is used to update the plain-text `code` directories (§1.1.1, p1).

The AIPS++ `makefiles` (p162) only ever refer to the RCS repositories indirectly, through a symbolic link called `rcs` which usually points to `slave` but could, in principle, be switched to point to `master` or `archive`.

```
         :
         :
         +- (master) --+...
         +- (archive) -+...
```

```
         |                          +--- apps ----+...
         |                          +- implement -+...
         |              +--- aips ----+-- fortran --+...
         |              |             +-- scripts --
         +---- rcs      |             +--- data ----
         |      =       |
/aips++ -+--- slave ---+
         |              +--- dish ----+...
         |              +- synthesis -+...
         |              +--- vlbi ----+...
         :              +-- contrib --+...
         :              :
```

inhale (p78) creates a `tmp` directory beneath the `slave` directory for the duration of its operation.

1.1.5 *ftp* directories

The AIPS++ ***ftp*** directories contain compressed ***tar*** files for distribution of AIPS++ sources. These reside directly under the AIPS++ master's root directory which on the AIPS++ master host *aips2.nrao.edu* is also the home directory of the anonymous ***ftp*** account, `~ftp/pub`. This allows the ***ftp*** directories, the master RCS repository, and the archive RCS repository to be accessible via anonymous ***ftp***.

```
         +-- master ---+...
         +-- archive --+...
~ftp ----+
         |              +-- master ---
         +---- pub ----+--- code ----
                        +-- import ---
```

There are three main ***ftp*** directories. The plain-text AIPS++ sources required for an end-user installation are stored in `pub/code`. Separate ***gzip***'d ***tar*** files are provided for each AIPS++ package (see §1.1.1, p1) in keeping with the optional nature of package installations.

Third-party public-domain software required by AIPS++ is redistributed in `pub/import`. If necessary, utilities essential for AIPS++ are automatically fetched from here by ***configure*** (p149) when AIPS++ is first installed.

The `pub/master` directory serves as a repository for the update files produced by the code distribution system (see ***exhale*** (p73)). ***exhale*** also creates a `tmp` subdirectory within it for the duration of its operation.

It is intended that ***ftp*** directories will also be provided for patches and distribution of binary installations once AIPS++ gains a user base. At that time consortium sites will also act as distributors of AIPS++ for their local geographic region, but until then the ***ftp*** distribution files will only be available from *aips2.nrao.edu*.

1.2 AIPS++ variable names

A standard set of variable names is used in AIPS++ scripts, makefiles and elsewhere to refer to AIPS++ directories (§1.1, p1). Except for `$AIPSPATH`, these are not stored in the environment but instead are rederived from `$AIPSPATH` whenever needed.

These variable names are freely used in this manual with the following typographic conventions:

`AIPSPATH` ...the name of the variable.

`$AIPSPATH` ...the value of the `AIPSPATH` variable, whether used as an environment variable or makefile variable.

`$(AIPSPATH)` ...the value of the `AIPSPATH` variable used as a makefile variable.

Additional variables used by the AIPS++ makefiles are described in the entries for `makedefs` (p155) and `makefiles` (p162).

```
ROOT = the AIPS++ home directory, also referred to as ~aips++
ARCH = the machine architecture, e.g. sun4, convex, etc.
SITE = the local site name
HOST = the host name

AIPSPATH = $ROOT $ARCH $SITE $HOST

AIPSROOT = $ROOT
AIPSARCH = $ROOT/$ARCH
AIPSSITE = $ROOT/$ARCH/$SITE
AIPSHOST = $ROOT/$ARCH/$SITE/$HOST

AIPSMSTR = $AIPSROOT/master
AIPSLAVE = $AIPSROOT/slave
AIPSRCS  = $AIPSROOT/rcs
AIPSCODE = $AIPSROOT/code
AIPSDOCS = $AIPSROOT/docs

MSTRETCD = $AIPSMSTR/etc

CODEINSD = $AIPSCODE/install
CODEINCD = $AIPSCODE/include

INSTARCH = $CODEINSD/$ARCH

LIBDBGD  = $AIPSARCH/libdbg
LIBOPTD  = $AIPSARCH/lib
BINDBGD  = $AIPSARCH/bindbg
BINOPTD  = $AIPSARCH/bin
BINTESTD = $AIPSARCH/bintest
```

```
ARCHTMPD = $AIPSARCH/tmp
ARCHAUXD = $AIPSARCH/aux
ARCHBIND = $AIPSARCH/bin
ARCHDOCD = $AIPSARCH/doc
ARCHMAN1 = $ARCHDOCD/man1
```

1.3 Configuration files

Two sets of configuration files are used in AIPS++, the `aipsrc` and `makedefs` files.

The `aipsrc` files store `keyword:value` entries used by AIPS++ scripts and programs. The mechanism is superficially similar to that of `.Xdefaults` on which it is modelled. The `aipsrc` files are hierarchical - a user's `.aipsrc` is usually the one first consulted, if no keyword definition is found therein the search continues to host-specific, site-specific, and default `aipsrc` (p38) files.

The `makedefs` files are GNU makefiles which define installation-specific variables. All AIPS++ `makefiles` (p162) begin by including `$AIPSARCH/makedefs` (p155) (§1.2, p8) which contains default definitions and also some generally applicable rules and targets. This "default" `makedefs` in turn includes `$AIPSSITE/makedefs` which allows the default definitions to be overridden.

1.4 AIPS++ accounts and groups

A set of AIPS++ accounts and groups have been defined to perform particular functions. The following descriptions include the standard names, user ids, and group ids, but different names may be defined via the `aipsrc` (p38) mechanism:

- *aips2adm* (uid=31414, gid=31414): Owns the master AIPS++ directories, RCS version files and ***ftp*** distribution files. *aips2adm* is responsible for maintaining the ***ftp*** directories, and in particular, runs the ***exhale*** (p73) ***cron*** job on the master host *aips2.nrao.edu*. *aips2adm*'s other functions are described in chapter 13. This account usually need only be created in Socorro.

- *aips2mgr* (uid=31415, gid=31415): Owns an AIPS++ installation, including directories, source files and binaries. At development sites, *aips2mgr*, and those in the *aips2mgr* group, are RCS and ***ftp*** administrators for an AIPS++ installation. In particular, they can directly invoke the ***rcs*** command on the AIPS++ RCS files. *aips2mgr* has an active login account but this may be restricted to selected machines. *aips2mgr* runs the daily ***inhale*** (p78) ***cron*** job.

- *aips2prg* (uid=31416, gid=31416): A generic programmer account and group. In practice the account may not often be used but the *aips2prg* group lists those who have permission to check sources into AIPS++, or check them out with a write lock (see §9, p89); there are no restrictions on checking sources out without a write lock.

- *aips2usr* (uid=31417, gid=31417): The *aips2usr* account is intended to serve as a generic AIPS++ usage account. The *aips2usr* group could be used to control access to AIPS++ executables and AIPS++ data areas if this degree of control was desired by the AIPS++

manager (the alternative being to give world execute and/or write permission). Neither the account nor the group are currently recognized by the system.

Chapter 2

Installation

This chapter [1] describes how to install AIPS++.

The installation of end-user (production-line) and AIPS++ consortium (code-development) systems are sufficiently different that separate instructions are provided for each.

If you contemplate installing AIPS++ for the Linux operating system using the GNU "binutils" package (e.g. *ar*, *ld*), then you should be aware that there is a bug in handling large archive (`.a`) files in versions 2.7 (and earlier). You can find out what version of binutils you have by running *ar -V*. If it is the incorrect version, either upgrade to version 2.8 (or later) or contact *aips2-request@nrao.edu* for a patch (all that is required is to change the `#if 1` that occurs at about line 823 of `bfd/archive.c` to `#if 0`).

2.1 External Developer's Release

*N*ote: The developer's release is being redone as a set of RPM's. You may find the RPMs at ftp://ftp.cv.nrao.edu/casa/development/rpms.

This section describes how to create an installation for external developers, i.e. developers who are not AIPS++ onsortium members. This sort of AIPS++ installation differs from the regular consortium installation in that:

1. It is intended as a stand-alone installation which allows developers to keep the installation in sync with the master repositories, but does not allow checking code into the system.

2. It is easy to configure.

3. It is made available for the most common architectures. Though currently it is **only** available for linux.

The developers installation operates by using CVSup[2] to both bootstrap the installation and keep the documentation, RCS source code repository, and data repository in sync with the master repositories at the AIPS++ center.

[1] Last change: Id: Installation.tex,v 19.7 2004/11/09 18:48:31 jmcmulli Exp
[2] http://www.polstra.com/projects/freeware/CVSup/

2.1.1 Getting Started

The first step to installing the AIPS++ external developer's release is to install the required software upon which AIPS++ depends. This includes CVSup[3], RPFITS[4], CFITSIO[5], and PGPLOT[6]. AIPS++ distributes RPMs[7] RPMs built upon various releases of RedHat Linux which greatly simplify the installation of these packages for Linux systems which use RPM. The RPMs are built for specific RedHat releases and must be installed as *root*. If you're using RPM, you need to install:

- `cvsup-16.1e-2ds.i386.rpm`
- `rpfits-2.4-1ds.i386.rpm`
- `cfitsio-2.401-1ds.i386.rpm`
- `pgplot-5.2.0-9ds.i386.rpm`
- `pgplot-devel-5.2.0-9ds.i386.rpm`
- `pgplot-motif-5.2.0-9ds.i386.rpm`

In addition to these, you need to ensure that a few packages which are normally part of a Linux installation are installed:

- GNU make[8]
- either Open MOTIF[9] or LessTif[10] (*or just about any other Motif you might have installed*)
- LAPACK[11] and BLAS
- all of the standard header files, e.g. `glibc-devel`
- all of the X11 header files, e.g. `XFree86-devel`
- `tcl` and `tk`
- bison[12] and flex[13]

A good way to check is to do something like:

```
rpm -qa | egrep '^openmotif|^lesstif|^make|^glibc|^XFree|^lapack|^blas|^tcl|^tk|^bison|^flex'
```

[3] http://www.polstra.com/projects/freeware/CVSup/
[4] http://www.atnf.csiro.au/computing/software/rpfits.html
[5] http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html
[6] http://astro.caltech.edu/~tjp/pgplot/
[7] ftp://aips2.nrao.edu/pub/import/Linux/RedHat/
[8] http://www.gnu.org/software/make/
[9] http://www.opengroup.org/openmotif/
[10] http://www.lesstif.org/
[11] http://www.netlib.org/lapack/
[12] http://www.gnu.org/software/bison/
[13] http://www.gnu.org/software/flex/

If the output of this seems to list all requirements, then you are in good shape.

Finally, you need to have the gcc 2.95 variety of GNU gcc compiler installed. This is necessary because we have found precision problems with g77 in the 2.96 release, and we do not yet support the 3.1 release. To simplify this, RPMs for `gcc-2.95.3` are also available from the AIPS++ ftp site[14]. You need:

- `gcc+2-2.95.3-52ds.i386.rpm`
- `gpp+2-2.95.3-52ds.i386.rpm`
- `libgpp+2-2.95.3-52ds.i386.rpm`
- `gppshare+2-2.95.3-52ds.i386.rpm`
- `g77+2-2.95.3-52ds.i386.rpm`

These RPMs install into `/usr/gcc/2.95.3`, e.g. `/usr/gcc/2.95.3/bin/gcc`, and currently they are not relocatable. Otherwise, you can get gcc 2.95 from the GNU ftp site[15] and install it by hand.

The next step is to select a directory where you would like your AIPS++ installation to reside. For the examples in this section, `/usr/local/aips++` will be used; if you choose a differnt location, just substitute your path for `/usr/local/aips++`. So first create the directory for the initial CVSup bootstrap file, for example:

```
bash$ mkdir -p /usr/local/aips++/sup/files
```

With an editor, create a file in this directory named for the architecture being installed, e.g. `/usr/local/aips++/sup/files/linux`. In this file, put:

```
bootstrap release=linux host=aips2.nrao.edu base=/usr/local/aips++
```

If you chose a root directory other than `/usr/local/aips++` for your installation, you should substitute your directory in the line above. This file specifies which set of bootstrap files should be fetched.

The next step is to use the bootstrap specification file you just created to fetch the bootstrap files:

```
bash$ cd /usr/local/aips++
bash$ cvsup sup/files/linux &
```

A GUI should pop up, when it does you should only need to press the *go* button, i.e. the button with the little **green triangle** beside the *exit* button. This will fetch all of the AIPS++ bootstrap files for the given architecture..

The next step is to run the bootstrap `setup` script. This script not only takes care of configuration details, but it also fetches the AIPS++ source code:

[14] ftp://aips2.nrao.edu/pub/import/Linux/RedHat/
[15] ftp://ftp.gnu.org/pub/gnu/gcc/gcc-2.95.3.tar.gz

```
bash$ cd /usr/local/aips++
bash$ linux/setup
```

This script will first verify the AIPS++ root directory, and then it will commence with downloading the source code. This may take some time, so patience, as well as bandwidth, is required. After the initial installation of the source code, however, subsequent updates should only download source code modifications, **not** all of the source code.

Lastly, all you need to do is build AIPS++:

```
bash$ cd /usr/local/aips++/code
bash$ gmake allsys
```

2.1.2 Updating the Source Code

If you later wish to bring your source code up to date with the current version available from the master repository at the AIPS++ center, you can do this easily with CVSup and the ontrol scripts which the setup step generated:

```
bash$ cd /usr/local/aips++
bash$ cvsup sup/files/code
bash$ cd code
bash$ gmake allsys
```

This procedure can be repeated whenever you wish, and subsequent updates will only download those portions of the code repository which have been modified or added. It **will not** download everything again.

2.1.3 Updating the Data Repository

In addition to a successful build of the surce code, the AIPS++ data repository is required to run AIPS++. This repository contains all of the data files needed to run AIPS++ as well as some data files used for testing and demos.

To update the data repository, you use CVSup:

```
bash$ cd /usr/local/aips++
bash$ cvsup sup/files/data
```

This will create a local mirror of the AIPS++ data repository. This can be redone later to update the repository. Subsequent updates will only download those portions of the data repository which have been modified or added. It **will not** download everything again.

2.1.4 Updating the Documentation

If you would like to have a local copy of the documentation (which is recommended), you use CVSup to get the documentation as well:

```
bash$ cd /usr/local/aips++
bash$ cvsup sup/files/docs
```

This will create a local mirror of the AIPS++ documentation and can be used to keep the documentation up to date, as with the code and data repositories.

2.1.5 Final Notes

CVSup provides an efficient method for maintaining synchroized mirrors, and the AIPS++ external developer's release makes heavy use of it. All of the control files for CVSup are kept beneath `sup/files`. These files are simple and reasonably straight forward to understand if a bit of time is spent with the CVSup documentation[16]. In addition to the control files mentioned above, the initial setup script also creates a `sup/files/update` script. This CVSup control script can be used to update the code and data repositories as well as the documentation with one CVSup invocation. After that, all that remains is the `gmake allsys` to build AIPS++.

2.2 End-user AIPS++ installations

A set of linux RPMS is available from ftp://ftp.cv.nrao.edu/casa. Please be aware that we are providing only minimal user support at this time (February 2004).

2.3 Consortium AIPS++ installations

Consortium installations differ from end-user installations in having a local copy of the master CVS source code repositories. The local code tree is updated regularly by the AIPS++ code distribution system via a procedure called ***inhale*** (p78). Consortium installations also have a mechanism for checking sources out of, and in to, the master CVS repositories.

Step 1. Create AIPS++ accounts

Before starting a consortium installation the following accounts must be created by the unix system administrator with the specified user and group ids in `/etc/passwd` (see §1.4, p9):

```
aips2mgr   uid=31415   gid=31415
aips2prg   uid=31416   gid=31416
aips2usr   uid=31417   gid=31417
```

The user and group ids correspond with those of the master sources and although different ids may be used it is *highly desirable* that they match. The home directory for the accounts should be set to the root directory of the AIPS++ tree. This can be anything but `/aips++` is preferred and is assumed in the following examples.

The following groups must be created with the matching group id and membership (in `/etc/group`):

[16] http://www.polstra.com/projects/freeware/CVSup/faq.html

```
aips2mgr   gid=31415
aips2prg   gid=31416   aips2mgr
aips2usr   gid=31417   aips2mgr aips2prg
```

You should also add your account name and the names of any other local AIPS++ managers to the *aips2mgr* group membership list. Do not add everyone to the *aips2mgr* group, it grants permission to directly manipulate the RCS sources. You should also add the names of all local AIPS++ programmers to the *aips2prg* group. This will allow them to check out and modify the AIPS++ sources. The *aips2mgr* account and the *aips2mgr* and *aips2prg* groups will be used during the installation.

Now create the AIPS++ root directory:

```
yourhost% mkdir /aips++
yourhost% chown aips2mgr /aips++
yourhost% chgrp aips2prg /aips++
yourhost% chmod ug=rwx,o=rx,g+s /aips++
```

In practice `/aips++` will often be a self-contained filesystem, usually on a separate disk. Allow 1 Gbyte of disk space for the AIPS++ system; any short-term surplus may be used for programmer workspaces.

This is as much as needs doing by the system administrator at this stage. The remainder of the initial part of the installation can be done by *aips2mgr*.

Step 2. Fetch the Source Code

First fetch the source code via anonymous CVS:

```
yourhost% cd /aips++
yourhost% cvs -r -d :pserver:anonymous@polaris.cv.nrao.edu:/home/cvs checkout -d code casa/code
```

This will fetch the main development branch of AIPS++. After the checkout completes you will need to configure the source installation

```
yourhost% cd code
yourhost% chmod 544 install/configure.cvs
yourhost% install/configure.cvs
```

You will be asked a series of questions, most of which have sensible defaults, aimed at constructing the AIPS++ `aipshosts` (p35) file. You then have to edit your site-specific `aipsrc` (p38) and `makedefs` (p155) files. Template versions of these files are supplied by ***configure*** (p149) and you should read the instructions carefully. After making your site-specific definitions ***configure*** will run some tests to check whether your `makedefs` definitions look sensible. Your `install` directory will then be made. This consists of a few C compilations and installation of some shell scripts, the most important of which is ***inhale*** (p78) itself. Ignore any error message from ***gmake*** (p154) concerning the non-existence of various subdirectories of `/aips++/code`.

Step 3. Run *sneeze*

At this point your AIPS++ installation has been bootstrapped to a state where ***sneeze*** (p85) can be run.

You should also have a C++ compiler, and a TEX installation which includes LATEX, ***dvips***, METAFONT and LATEX2HTML. Unset the `DOCSYS` variable in `makedefs` if you don't have TEX, it will prevent compilation of the AIPS++ documentation. The documentation may be downloaded using the ***aupdate*** command.

Users of SysV based systems such as Solaris should be warned that ***inhale*** (p78) requires the BSD version of ***sum*** for computing checksums. You must ensure that the BSD version will be found ahead of the SysV version in *aips2mgr*'s `PATH`. The GNU version of ***sum*** (in the GNU "fileutils" kit) provides both algorithms and uses BSD by default. Less salubrious possibilities are to put `/usr/ucb` (Solaris) or `/usr/bsd` (IRIX) ahead of `/usr/bin` in *aips2mgr*'s `PATH`, or to create a symlink to the BSD version of ***sum*** in the AIPS++ `bin` area.

First invoke ***aipsinit*** (p31) to add the AIPS++ `bin` directory to your `PATH`. If your interactive shell is a C-like shell (***csh***, ***tcsh***) you would use

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

whereas for Bourne-like shells (***sh***, ***bash***, ***ksh***) you would use

```
yourhost% . /aips++/aipsinit.sh
```

If you use some other shell you'll have to revert to one of the above for the remainder of the installation. Now invoke ***sneeze*** (p85)

```
yourhost% sneeze -l -m cumulative&
```

This will build and install the latest version of the sources which are under active development. If you made any mistakes in your `aipsrc` or `makedefs` definitions some of these may become apparent during the installations. After fixing them you can recover via

```
yourhost% gmake -C /aips++/code allsys
```

The `allsys` target will compile all AIPS++ sources, including documentation (assuming of course that you have the compilers). If you just wanted to compile the documentation alone you could use

```
yourhost% gmake -C /aips++/code docsys
```

If everything has gone properly you should now have an up-to-date AIPS++ installation. However, in order to keep it up-to-date you must define a ***cron*** job to run ***inhale*** (p78) on a regular basis. The normal procedure is to do a cumulative update every Saturday evening. However, you may wish to maintain a (possibly separate) system which is updated on a daily basis. Half-daily updates are also possible, but note that 12 hours may not be sufficient time to rebuild the system.

The exact timing depends on your timezone with respect to the master. New updates are produced by 0700 and 1900 Socorro time (MST or MDT) but you should allow at least an hour's grace before collecting them. An example `crontab` file might resemble the following:

```
# Cumulative update of the AIPS++ directories each Saturday evening.
00 22 * * 6   (. $HOME/.profile ; inhale.cvs -c) 2>&1 | \
   mail aips2mgr aips2-inhale@nrao.edu
```

(Note that all ***cron*** entries must be one-liners but they are broken here for clarity.) You may need to add the ***-n*** option to ***inhale*** (p78) accordingly. Note that, as in the above example, the log produced by ***inhale*** is generally forwarded to *aips2-inhale@nrao.edu*. These logs are archived for about 10 days and are accessible via the AIPS++ home page *http://aips2.nrao.edu/aips++/docs/html/aips++.html*. This is particularly useful for verifying code portability, especially on platforms that a programmer doesn't have ready access to. You should also add the email address of a local person who will monitor the ***inhale*** logs (*aips2mgr* in the above example).

Step 4. CVS access to the development tree

For active developers, you will need a CVS account on the main CVS repository server. Please send an encyrted password to aips2mgr@aoc.nrao.edu. For more information about CVS visit the webpage at http://www.cvshome.org.

Chapter 3

Setting up

This chapter [1] describes how to set up your AIPS++ environment.

3.1 Setting up your account to use AIPS++

3.1.1 General AIPS++ setup

The AIPS++ environment is defined via a once-only modification to your shell's startup script.

Assuming that AIPS++ has been installed under `/aips++`, users of Bourne-like shells (*sh*, *ksh*, *bash*) must add the following to their `.profile` file at a point *after* `PATH` (and `MANPATH`) are defined:

```
# Get the AIPS++ environment.
  [ -f /aips++/aipsinit.sh ] && . /aips++/aipsinit.sh
```

The equivalent entry in `.login` for C-like shells (*csh*, *tcsh*) is:

```
# Get the AIPS++ environment.
  if (-f /aips++/aipsinit.csh) source /aips++/aipsinit.csh
```

Users of the *rc* shell might use

```
# Get the AIPS++ environment.
  if (test -f /aips++/aipsinit.rc) . /aips++/aipsinit.rc
```

in `.rcrc`, or in `.esrc` for the *es* shell:

```
# Get the AIPS++ environment.
  if {test -f /aips++/aipsinit.es} {. /aips++/aipsinit.es}
```

[1] Last change: Id: SettingUp.tex,v 19.0 2003/07/16 04:18:28 aips2adm Exp

The ***aipsinit*** (p31) scripts define a single environment variable called `AIPSPATH` (§1.2, p8) and add the AIPS++ `bin` area to the `PATH` environment variable and the AIPS++ `man` directory to the `MANPATH` environment variable.

For more detailed information, see ***aipsinit*** (p31). This contains an explanation of `AIPSPATH` and a mechanism for controlling the point where the AIPS++ `bin` areas are added to `PATH`.

3.1.2 AIPS++ programmer setup

Normally, AIPS++ programmers must belong to the AIPS++ programmer group in order to have permission to write to the AIPS++ source directories. The conventional name for this group is *aips2prg* (see §1.4, p9), but it may be different at your site. Type

```
yourhost% groups
```

to list all groups that you are a member of, and consult your local AIPS++ manager if in doubt.

Apart from invoking ***aipsinit*** (p31), AIPS++ programmers need to create a shadow copy of the AIPS++ `code` directory tree to serve as their AIPS++ workspace. The ***mktree*** (p126) utility does this:

```
yourhost% mkdir $HOME/aips++
yourhost% cd $HOME/aips++
yourhost% mktree
```

Apart from creating a shadow copy of the AIPS++ `code` directory tree, ***mktree*** creates symbolic links into the AIPS++ `rcs` directory tree thereby linking the programmer's workspace to the local copy of the AIPS++ RCS repository (see §1.1.4, p5).

mktree works incrementally so that if any workspace directories or `RCS` symbolic links are accidently deleted, or if new AIPS++ directories are created, ***mktree*** will recreate only what is necessary. For a more detailed description, see ***mktree*** (p126).

Once the programmer workspace has been created by ***mktree*** AIPS++ sources can be checked in and out, updated, renamed, or deleted by the ***ai*** (p93), ***ao*** (p97), ***au*** (p109), ***amv*** (p105), and ***ax*** (p115) utilities. Native RCS utilities such as ***rlog*** and ***rcsdiff*** may also be used. For a more detailed description see chapter 9.

Chapter 4

Data Repository

This chapter [1] describes how to install and contribute to the AIPS++ data repository.

The AIPS++ data repository contains all of the standard *global* data used internally by AIPS++. This data was separated from the main code tree because it greatly increased the volume of the AIPS++ source distributions.

The intended audience for this repository includes both AIPS++ developers who use data to implement their applications and users who access standard catalogs and images which are components of the repository. The data is typically provided as AIPS++ tables, but some data is provided as FITS files. Other data formats may also be included for the purposes of testing conversion procedures, e.g. filling non-FITS correlator data.

In the future, the goal is that components of the repository can be automatically mirrored to the user's personal workspace when they are not available in the the local AIPS++ installation. This will minimize the amount of data which all sites must keep locally while making the entire repository available to users.

4.1 Mirroring the Repository

The AIPS++ data repository is generated at a central site and is then mirrored by all consortium and user sites. This allows the central site to build and maintain the data, and it allows user sites to easily stay in sync with the master repository. In fact, it is easy for these secondary sites to act as mirrors from which tertiary sites can retrieve the data.

The first step to retrieving the data repository is to install CVSup[2]. While CVSup is optimized for mirroring CVS[3] repositories, it can mirror arbitrary collections of files. CVSup is the mirroring tool used by FreeBSD[4]. CVSup was chosen because it excels at minimizing the information which must be transferred by incrementally updating files using whatever information is known about individual files.

[1] Last change: Id: Data.tex,v 19.0 2003/07/16 04:18:31 aips2adm Exp

[2] http://www.polstra.com/projects/freeware/CVSup/

[3] http://www.cyclic.com/

[4] http://www.freebsd.org/

It is easiest to just install CVSup binaries because CVSup is written in Modula-3[5]. CVSup binaries for many architectures can be found at the FreeBSD FTP site[6]. You can choose between versions which include or exclude the GUI. However, the GUI version is recommended since the GUI is nice and can be disabled with a command line flag.

Once you have installed CVSup, you are ready to create a supfile which describes how to retrieve and install the data repository. First log in as *aips2mgr* and create a directory where the repository should be mirrored, e.g. `/aips++/data`. It is fine for many AIPS++ installations to share the same data repository. So if you have more than one AIPS++ installation, you will likely want to mirror the repository in a central location and then just create symbolic links from the AIPS++ installations to the repository. If you have only one AIPS++ installation, then calling the directory `data` and putting it in the root of the AIPS++ installation (at the same level as `code`) is the best choice.

Next edit a file called `supfile` in the directory just created. To retrieve the whole repository, which currently isn't so large, you would put the following in the supfile:

```
*default host=aips2.nrao.edu compress
*default base=YOUR_DIRECTORY_HERE
*default release=all
*default delete use-rel-suffix
data
```

You should replace `YOUR_DIRECTORY_HERE` in the `base` assignment with the fully qualified path to the directory you just created to contain the repository.

After you have created the supfile, you are ready to install the repository. Just run:

```
yourhost% cvsup supfile
```

If you don't run this from the newly created directory, you will have to fully qualify the path to the supfile for ***cvsup***. This should run and create a mirror of the data provided on the central AIPS++ server.

If you want to continue to keep your data repository in sync with the repository on the master AIPS++ server, you need to create a ***cron*** job entry to run ***cvsup*** regularly. This entry might look like:

```
00 6 * * 0 /usr/local/bin/cvsup -g /aips++/data/supfile 2>&1 | mail aips2mgr
```

This would update the data repository once per week on Sunday at 6AM. It is best to synchronize the updates of the data repository with the update of local AIPS++ installation.

If you have multiple AIPS++ installations which will be sharing a single data repository, create a symbolic link called `data` in the root of each installation (at the same level as `code`) to the data repository.

[5]http://www.research.digital.com/SRC/modula-3/html/home.html
[6]ftp://ftp.FreeBSD.org/pub/FreeBSD/development/CVSup/binaries/

4.2 Repository Structure

Data in the repository is divided based on content. The top layer of the repository divides the data into broad categories:

catalogs/lines spectral line catalogs, e.g. the Poynter and Pickett *Submillimeter, Millimeter, and Microwave Spectral Line Catalogue.*

catalogs/sources pulsar, continuum, and spectral line source catalogs.

geodetic data related to the earth, e.g. global positions, orientation, magnetic field, etc.

ephemerides data related to the movement of planets, comets, and satellites.

demo data used in tests and demos of AIPS++.

nrao data specific to nrao facilities.

other observatory data specific to an *other observatory*.

Beneath this broad categorization is the actual data which is used by AIPS++ applications.

4.3 Contributing to the Repository

This section describes how to setup and use CVS to contribute to the repository. While it introduces the basic CVS commands, it is not intended as a substitute for existing CVS documentation, e.g. the CVS support page[7] at *cyclic.com*. Standard CVS documentation should be consulted before attempting to contribute to the repository for the first time.

Setup CVS

It is more difficult to setup things to contribute to the AIPS++ data repository than to mirror it. The first requirement is that the user must have an account on the AIPS++ central server, *aips2.nrao.edu*, and the user id on *aips2.nrao.edu* must be a member of the *aips2prg* group. In addition, CVS and SSH must be installed on your local system. These packages are widely available.

With ***ssh*** you create your pair of keys using ***ssh-keygen***. This will create `~/.ssh/identity.pub` on your local machine (or `~/.ssh/id_dsa.pub` for ssh2 based versions) You then add the contents of this file to `~/.ssh/authorized_keys` (`~/.ssh/authorized_keys2` for ssh2 based) on *aips2.nrao.edu*.Make sure `authorized_keys` is readable only by the user. After that, you are ready to try ***ssh***; try something like:

```
yourhost% ssh aips2.nrao.edu which cvs
yourhost% ssh aips2.nrao.edu which co
```

[7] http://www.cyclic.com/CVS/support

from your local machine. You must get these to return the path to the *cvs* and *co* binaries before going further. These binaries are located in `/opt/local/gnu/bin` on *aips2.nrao.edu*. To make this path available, you may need to edit `~/.ssh/environment` on *aips2.nrao.edu*, and add a line like:

```
PATH=/usr/bin:/opt/local/bin:/opt/local/gnu/bin
```

Shell variables, e.g. `$PATH`, are not expanded.

Once you have resolved all of the problems involved with running remote commands on *aips2.nrao.edu*, you are ready to check out a piece of the data repository. First you must tell your local *cvs* installation to go to *aips2.nrao.edu* for the repository and to use *ssh* instead of *rsh* to run remote commands. For the Bourne-like shells, you do this like:

```
yourhost% CVS_RSH=ssh
yourhost% CVSROOT=:ext:aips2.nrao.edu:/aips++/cvs
yourhost% export CVS_RSH CVSROOT
```

for C-like shells, you would do:

```
yourhost% setenv CVS_RSH ssh
yourhost% setenv CVSROOT :ext:aips2.nrao.edu:/aips++/cvs
```

Now to check out a piece of the source from which the data repository is built, you would do something like:

```
yourhost% cvs co data/nrao
```

This would check out the data specific to NRAO. Unlike RCS, when you check out files from CVS the files are not locked. Other users can check out and check in changes to the same files. Checking out the files just provides a copy of the current state of the files to work with. Any conflicts between your changes and changes which have occurred since you checked out the files are addressed at check in time (actually *cvs update* time).

Using CVS

The format for running CVS commands is:

```
yourhost% cvs <COMMAND> <OPTIONS>
```

As shown above, the command for checking out code is `co` (abbreviation for `checkout`). The command for adding new files or directories is `add`. The command for checking in changes to files is `commit`. `diff` is used to print the differences between a local copy and the version resident in the repository.

All commands in CVS are recursive by default. So if you wanted to list all of your changes, you would go to the top of the tree and run:

```
yourhost% cvs diff .
```

which says give me the difference in this directory including every directory beneath this directory.

When you check out files, you can check out all of the repository source:

```
yourhost% cvs co data
```

or only a small portion:

```
yourhost% cvs co data/geodetic/IGRF
```

After you have had code checked out for some time, you may want to bring it up to date with the current state of the code in the CVS repository. You use `update` to do this, e.g.

```
yourhost% cvs update .
```

`update` will not trample over your changes. It updates all files files which have not been modified, and where possible it merges in differences to files which you have modified without any problems. Where conflicts occur, it includes your changes and other changes in a *diff-like* format. When you get a warning about this problem, you must edit the file by hand and reconcile your changes and the other changes.

For more information about CVS, see the support page[8] at *cyclic.com*.

CVS Example

Checking out a portion of the data repository is the first step toward contributing to the repository. Whether you plan to modify existing data, check-in new data, or remove data, you must first checkout that portion of the repository which you plan to change. As an example, lets assume that we want to check in test data for a new AIPS++ package, *foobar*. The data we are adding is a table, *footest1*. So we first checkout the `demo` portion of the data repository:

```
yourhost% cvs co data/demo
```

This creates a workspace where we can make changes.

Next since *foobar* tests will need multiple data sets, we want to create the directory to group these data sets:

```
yourhost% cd data/demo
yourhost% mkdir foobar
yourhost% cvs add foobar
```

[8] http://www.cyclic.com/CVS/support

As discussed in section §4.4, p27, makefile rules for installing tables from the data repository source tree to the repository tree which is exported to end users depend on the actual table being below a directory of the same name. In our case, the tree looks like:

```
data                              workspace root directory
|
+- demo                           all test and demo data
   |
   +- foobar                      all of our foobar test data
      |
      +- footest1                 makefile to install footest1
         |
         +- footest1              actual footest1 data
```

This duplication of the table (*footest1*) is done because unlike our simple example some tables are automatically updated with scripts run by the makefiles found in the first level directory, i.e. there is no lower level directory in the data repository source tree.

So we create a *footest1* directory which will contain the actual table as well as a makefile to install our table in the export area.

```
yourhost% cd foobar
yourhost% mkdir footest1
yourhost% cvs add footest1
```

We also create a simple makefile which will install our table from the repository source tree to the export tree. It looks like:

```
DATA_THISDIR  := $(shell pwd | sed -e 's=^/tmp_mnt/=/=')
DATA_ROOT := $(shell echo $(DATA_THISDIR)/ | sed -e '{s@/data/.*$$@/data@;}')
include $(DATA_ROOT)/config/makedefs/basic
include $(DATA_ROOT)/config/makedefs/install.table
```

This is also discussed in section §4.4, p27. In the *footest1* directory just created, we check in this makefile and our table (which we copy for our work area):

```
yourhost% cd footest1
yourhost% cp ../../glishtutorial/t1/makefile .
yourhost% cvs add makefile
yourhost% mkdir footest1
yourhost% cvs add footest1
yourhost% cd footest1
yourhost% cp ~/footest1/table.* .
yourhost% cvs add table.*
```

Here, the standard table makefile was copied from a table already in the repository, and the table contents were copied from elsewhere, i.e. `~/footest1`.

Using `add` schedules the files for addition to the repository, but none are actually added until we do a `commit`:

```
yourhost% cd ../../..
yourhost% cvs commit -m 'initial check-in of foobar' foobar
```

The `cd` gets us to the demo directory, the `-m` on the `commit` line supplies the log message and the `foobar` at the end of the line indicates that the `foobar` subdirectory and all of its contents should be committed to the repository. After all of this completes successfully, we are through with this workspace (created with the initial `cvs co`). It can be deleted if it is no longer needed.

4.4 Repository Conventions

There are certain conventions which must be followed when contributing to the repository. In general, the makefiles which are used to build the data repository are much simpler, by design, then those used to build the AIPS++ system. This is possible because the data repository makefiles only build the repository on one system and the work they must do is much more limited than that of the AIPS++ makefiles.

Organization

All of the scripts, makefile includes, etc. which are used to build the data are kept beneath `config` directories. The directories which may be found beneath `config` are `bin` for executable scripts, `libexec` for scripts which are included rather than executed directly, and `makedefs` for the small bundles of functionality which are included by other makefiles to allow them to do their work. A `config` tree can occur anywhere in the data repository source tree. The principal is that the `config` files should be placed as far down on the source tree as possible. For example, a script would only be placed in the top level `data/config/libexec` directory if it is included by scripts in more than one of the broad top level categories, e.g. `measures_util.g` is used in both `ephemerides` and `geodetic` so it must be placed in this top level config directory. If it were used only in `geodetic`, it would be moved down to the `data/geodetic/config/libexec` directory. The reason for this rule is so that wherever possible checking out the lowest level `config` directory along with the top `config` directory and the source for a particular data component of the repository will be sufficient to build that particular component.

Using Scripts

In many cases, an AIPS++ table is simply copied from the data source tree to the repository. There are, however, instances where an AIPS++ table is created (and updated) from files retrieved from remote sites. During the build, all of the tools available in an AIPS++ installation will be available for the makefiles and scripts to use. However, the plan is that no compilation, e.g. of C++ code, should be necessary to build the data repository. It should all be done with scripts. Currently, it is done with Glish scripts. Should there be cases where this is impossible this rule can be revisited.

When using scripts which build a piece of data, the exit status of the script indicates the result. In particular, an exit status of `0` indicates that the script completed successfully and that the data item *was* updated, an exit status of `1` indicates that the script completed successfully but the data item *was not* updated, and finally, an exit status of `-1` indicates that an error occurred during the execution of the script. The exit status of the scripts is used to generate logs containing only a list of the updates or errors which can then be used to keep tabs on the system without getting daily email messages.

Makefiles

In general, there is a one-to-one mapping between directories in the data source tree and directories in the data repository tree. The one slight deviation from this general rule is with tables. For tables, there is a replication of the table directory. So for example if we have a table called `demo/3C273XC1` in the repository tree, this table would actually correspond to `demo/3C273XC1/3C273XC1` in the source tree. The makefile, scripts (which need not be beneath a `config/libexec` directory), text files, etc. which install and perhaps create this table are found in the first `3C273XC1` directory while the `3C273XC1` subdirectory only contains the table which is installed in the repository.

When writing makefiles, there are some basic tools which should be used. `$(RUN)` *must* be used to execute the scripts used to generate a data component from scratch. This collects the exit status of the script and then uses the status to produce the update and error logs. Another tool which is often useful is `$(MKHIER)`. When a fully qualified path is passed in, this will create the entire path if it doesn't exist, e.g. a makefile rule like:

```
doit:
        $(MKHIER) /tmp/just/trying/it/out
```

would cause all of the directories in `/tmp/just/trying/it/out` to be created.

In addition to these tools, there are some variables which are useful:

$(DATA_ROOT) fully qualified path to the root of the data tree, *set in each makefile.*

$(DATA_THISDIR) fully qualified path to the current directory, *set in each makefile.*

$(DATA_INSTALLTHISDIR) the install point in the data repository which corresponds to the current directory.

$(AIPSROOT) fully qualified path to the root of the AIPS++ installation available for use in building the data repository.

$(AIPSARCH) fully qualified path to the architecture directory of the AIPS++ installation.

When writing makefiles in the data system, there are two targets which are recursively built these are `build` and `install`. `build` is the target which builds data elements from scratch.

`install` is the target which moves the data element from the source tree to the mirror point for the repository. Obviously all makefiles will specify an `install` target, but not all makefiles will specify an `build` target.

Much of the standard work done by the makefiles is found in *makedef* components which are included in the makefiles for individual data elements. All makefiles include a standard preample which does the minimum setup required to find the *makedef* include files:

```
DATA_THISDIR  := $(shell pwd | sed -e 's=^/tmp_mnt/=/=')
DATA_ROOT := $(shell echo $(DATA_THISDIR)/ | sed -e '{s@/data/.*$$@/data@;}')
include $(DATA_ROOT)/config/makedefs/basic
```

All makefiles should simply include this at the beginning. The `basic` makedef sets up all of the standard variables and rules for recursively building the `build` and `install` targets. Including the `install.table` makedef defines the rule which will copy a table from the current directory to the repository. It is assumed that the table is beneath the current directory and has the same name as the current directory. Including the `install.fits` makedef defines the rule which will copy all FITS files in the current directory to the corresponding directory in the repository. By convention, **all** FITS files have the `.fits` suffix.

So for example, the makefile to just copy a table which is not updated as part of the `build` process to the repository would look like:

```
DATA_THISDIR  := $(shell pwd | sed -e 's=^/tmp_mnt/=/=')
DATA_ROOT := $(shell echo $(DATA_THISDIR)/ | sed -e '{s@/data/.*$$@/data@;}')
include $(DATA_ROOT)/config/makedefs/basic
include $(DATA_ROOT)/config/makedefs/install.table
```

The makefile to install FITS files would look like:

```
DATA_THISDIR  := $(shell pwd | sed -e 's=^/tmp_mnt/=/=')
DATA_ROOT := $(shell echo $(DATA_THISDIR)/ | sed -e '{s@/data/.*$$@/data@;}')
include $(DATA_ROOT)/config/makedefs/basic
include $(DATA_ROOT)/config/makedefs/install.fits
```

and the makefile to build and install a table might look like:

```
DATA_THISDIR  := $(shell pwd | sed -e 's=^/tmp_mnt/=/=')
DATA_ROOT := $(shell echo $(DATA_THISDIR)/ | sed -e '{s@/data/.*$$@/data@;}')
include $(DATA_ROOT)/config/makedefs/basic

build:
        @$(RUN) glish IERSeop97.g

include $(DATA_ROOT)/config/makedefs/install.table
```

A good example of how to create and update tables can be found in the scripts and makefiles beneath `data/geodetic/IERSeop97`. In this case, Glish is used to create and maintain the `IERSeop97` table.

Chapter 5

Environment

Utilities and databases which allow AIPS++ users and programmers to define their AIPS++ environment [1].

5.1 *aipsinit*

AIPS++ initialization procedure.

Synopsis

`. aipsinit.sh`	...for Bourne-like shells, e.g. *sh*, *bash*, *ksh*
`source aipsinit.csh`	...for C-like shells, e.g. *csh*, *tcsh*
`. aipsinit.rc`	...for the *rc* shell
`. aipsinit.es`	...for the *es* shell

Description

The *aipsinit* scripts define the `$AIPSPATH` environment variable required by AIPS++, and modify the `PATH` environment variable (and `MANPATH` if defined) to include the relevant AIPS++ `bin` (and `man`) directories. They are normally invoked at login by the `.profile` or `.login` files.

The `$AIPSPATH` environment variable consists of a sequence of four character strings separated by spaces containing the following information:

1. Root of the AIPS++ directory tree.
2. Host "architecture".
3. Local site name.
4. Host name.

[1] Last change: Id: Environment.tex,v 19.0 2003/07/16 04:18:26 aips2adm Exp

aipsinit modifies the PATH environment variable to include the AIPS++ binary directories as follows:

1. Firstly, any prior AIPS++ path definition is either removed, or if $AIPSPATH was already defined, translated back into an aips_bin keyword.
2. If the keyword aips_bin is encoded into PATH it is replaced with the fully qualified pathname.
3. If the aips_bin keyword is not present, the AIPS++ bin directory is added to the start of PATH, following "." if it came first.

The keyword mechanism allows full control over placement of the AIPS++ bin directories in PATH and is the recommended method.

If $AIPSPATH is already defined when *aipsinit* is invoked it will also check for any AIPS++ programmer directories in the PATH, for example any referring to programmer bin directories, and if so will attempt to amend them. This works only if the programmer workspace is rooted at a directory named aips++.

If the MANPATH environment variable is defined, *aipsinit* modifies it in the same manner as for PATH except using the aips_doc keyword. If the aips_doc keyword is not present, the AIPS++ man area is always added to the start of MANPATH.

The host "architecture", a string of the form <base>[_<ext>], corresponds to a subdirectory of the AIPS++ root directory in which an AIPS++ system resides (object libraries, executables,...). The base part of the name is a mnemonic predefined by AIPS++ which describes the binary format of a machine and reflects the combination of machine architecture and operating system. The optional extension is used to select between alternative systems of the same binary format, for example as may have been generated by different compilers.

aipsinit recognizes an environment variable (or shell variable in aipsinit.csh) aips_ext which, if defined, overrides the architecture extension defined in the aipshosts (p35) file for a particular machine. By convention, setting aips_ext to "_" signals no extension, that is, any architecture extension specified in aipshosts is stripped off. For certain shells (*bash*, *csh*, *es*, *ksh*, *rc*, and *tcsh*) *aipsinit* also defines an alias or shell function also called *aipsinit* which takes a value for aips_ext as an argument and reinvokes *aipsinit*. This provides a convenient mechanism for interactively switching between architecture extensions. Note that *aipsinit* always undefines aips_ext.

Options

The *aipsinit* scripts take no options directly (such scripts cannot take arguments). However, see the description of the aips_bin and aips_doc keywords above, and also the aips_ext environment (or shell) variable.

Resources

The information for setting the architecture and site name is contained in the aipshosts (p35) file. If this does not contain an entry for a particular machine, *aipsinit* tries to deduce it.

Examples

For Bourne-like shells the following sequence might be used in `.profile`:

```
PATH=".:$HOME/bin:aips_bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin"
MANPATH=".:$HOME/man:aips_doc:/usr/local/man:/usr/man"
[ -f /aips++/aipsinit.sh ] && . /aips++/aipsinit.sh
```

For C-like shells the following sequence might be used in `.login`:

```
setenv PATH ".:~/bin:aips_bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin"
setenv MANPATH ".:~/man:aips_doc:/usr/local/man:/usr/man"
if (-f /aips++/aipsinit.csh) source /aips++/aipsinit.csh
```

or alternatively

```
set path = (. ~/bin aips_bin /usr/local/bin /usr/ucb /usr/bin /bin)
if (-f /aips++/aipsinit.csh) source /aips++/aipsinit.csh
```

For the *rc* shell the following sequence might be used in `.rcrc`:

```
PATH=.:$HOME/bin:aips_bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin
MANPATH=.:$HOME/man:aips_doc:/usr/local/man:/usr/man
if (test -f /aips++/aipsinit.rc) . /aips++/aipsinit.rc
```

For the *es* "extensible" shell the following sequence might be used in `.esrc`:

```
PATH=.:$HOME/bin:aips_bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin
MANPATH=.:$HOME/man:aips_doc:/usr/local/man:/usr/man
if {test -f /aips++/aipsinit.es} {. /aips++/aipsinit.es}
```

Files

`$AIPSROOT/aipshosts` ...database of AIPS++ hosts.

`$AIPSROOT/.login` ...*aips2mgr*'s `.login` file.

`$AIPSROOT/.profile` ...*aips2mgr*'s `.profile` file.

The `.login` (p45) and `.profile` (p47) files are provided for *aips2mgr*'s interactive and *cron* jobs.

See also

`aipshosts` (p35), AIPS++ hosts database.
`.login` (p45), *aips2mgr*'s `.login` file.
`.profile` (p47), *aips2mgr*'s `.profile` file.

Author

Original: 1992/03/05 by Mark Calabretta, ATNF.

5.2 aipshosts

AIPS++ hosts database.

Synopsis

`$AIPSROOT/aipshosts`

Description

The `aipshosts` database is an ASCII file containing a list of AIPS++ hosts at a site (or possibly at a set of sites within an institution). It is used by the ***aipsinit*** (p31) scripts in defining the `$AIPSPATH` environment variable.

`aipshosts` provides a means of specifying an architecture and site for a set of AIPS++ hosts. Its existence was prompted by the fact that there is no generic way to determine a machine's architecture. However, in addition to machine architectures, the architecture may differentiate between operating system versions or compilers where there is a choice of these at a particular site. Hosts from several sites within an institution may be included within one `aipshosts` file thereby making it possible to copy an AIPS++ installation verbatim from the central site to a remote site.

`aipshosts` is initially constructed by the ***configure*** (p149) script when AIPS++ is first installed. Thereafter it can be maintained manually by adding, deleting or modifying entries as required.

Each host has a one-line entry in the `aipshosts` file, the format of which is as follows:

1. The host name. A catch-all value "`DEFAULT`" applies to any host which does not have a specific entry. If the default is "`NONE`" then the aipsinit scripts will not setup `AIPSPATH` for any host not explicitly listed in `aipshosts`.
2. The host architecture. This specifies a subdirectory of `$AIPSROOT`, referred to as `$AIPSARCH` (§1.2, p8), in which the AIPS++ system for the machine resides. In particular, the `bin` and `doc` subdirectories of `$AIPSARCH` are encoded into the `PATH` and `MANPATH` environment variables by ***aipsinit*** (p31).

   The host "architecture" is a string of the form `<base>[_<ext>]`. The base part of the name describes the binary format of a machine and reflects the combination of machine architecture and operating system. Because some AIPS++ scripts may need to switch on architecture type, the base part of the architecture name has been standardized. Currently recognized values are:

   ```
   alpha      ...DEC/Alpha, OSF/1
   hpux       ...Hewlett-Packard, HPUX
   ibmrs      ...IBM RS6000, AIX
   sgi        ...Silicon Graphics, IRIX
   sun4       ...SUNsparc, SunOS 4.x
   sun4sol    ...SUNsparc, Solaris 2.x
   ```

An optional extension to the architecture type can be used to select between alternative systems of the same binary format, for example as may have been generated by different compilers, or different compiler options. Apart from not containing an "_" or ":", the extension may be freely chosen. However, in order to provide template `makedefs` (p155) and `aipsrc` (p38) files when AIPS++ is installed, ***configure*** (p149)) recognizes a number of standard combinations. Specifically

- The compiler supplied with the operating system (whether bundled or unbundled) is denoted by `ntv`.
- The GNU compiler (*g++*) is denoted by `gnu`.

Where it's desired to maintain several systems using different compiler options within the one AIPS++ installation (for example, different levels of optimization) this may be denoted by `O`n, where n is an optional digit. This should be placed *after* any compiler specification, separated by a "-", for example, `sun4sol_gnu-O2`. (Note that AIPS++ already supports separate debug and optimized compilations within each system.)

Any architecture extension defined in `aipshosts` serves only as the default for a particular machine. ***aipsinit*** (p31) provides a mechanism which allows it to be redefined on demand.

3. The site name. This specifies a subdirectory of the `$AIPSARCH` directory (§1.2, p8) containing the site-specific configuration files `aipsrc` (p38)) and `makedefs` (p155). The site name is chosen when AIPS++ is first installed and thereafter must be used consistently for all hosts at a site.

4. A comment which may usefully describe the machine and has no other purpose.

Comments within the `aipshosts` file are denoted by a `#` in column 1, and blank lines are ignored.

Note that when hosts are added to the system the host-specific system area may be created beneath `$AIPSSITE` (§1.2, p8) to hold host-specific databases (see `aipsrc` (p38)). Likewise, when a host is removed its system area should be deleted.

Examples

The following might be a fragment of an `aipshosts` file covering all of ATNF.

```
# AIPS++ host database

DEFAULT   sun4sol_ntv   epping
cetus     sun4sol_ntv   epping         Sun Ultra 1, AIPS++ server Room 85
raptor    sgi_gnu       epping         Silicon Graphics, computer room.
corvus    sun4sol_gnu   culgoora       Sun 4/65 SPARC 1+, control room.
yeti      alpha_gnu     parkes         AlphaServer 1000 4/266
```

Files

`$AIPSROOT/aipshosts` ...database of AIPS++ hosts.

See also

AIPS++ variable names (§1.2, p8).
aipsinit (p31), AIPS++ initialization procedure.
aipsrc (p38), AIPS++ resource database.
configure (p149), AIPS++ installation script.
makedefs (p155), AIPS++ makefile definitions.

5.3 aipsrc

AIPS++ resource database.

Synopsis

```
aipsrc
```

Description

The aipsrc files store keyword:value entries used by AIPS++ scripts and programs. The mechanism is similar to that of .Xdefaults on which it is modelled. The form of an aipsrc entry is

```
keyword: value
```

where the keyword, which must begin in column 1, usually consists of a sequence of character strings separated by asterisks (*) or periods (.), and terminated by a colon (:). The value consists of everything after the last whitespace character (space or tab) following the colon.

Keywords may use "*" as a wildcard. However, since the procedure which reads the aipsrc file accepts the first match it finds, the order of the entries may be significant. For example, consider the following two aipsrc entries

```
printer*paper:              A4
printer.dp1.paper:          A3
```

If a match was requested for the keyword printer.dp1.paper the first entry would be returned, contrary to what was probably wanted. In fact, the second entry would never be operative.

Comments in the aipsrc file have a # in column 1, and blank lines are ignored.

Resources

The following aipsrc files are consulted in sequence by *getrc* (p43) until a match is found (except that a user's /.aipsrc is ignored if the *-i* option is specified)

```
~/.aipsrc
$AIPSROOT/.aipsrc
$AIPSHOST/aipsrc
$AIPSSITE/aipsrc
$AIPSARCH/aipsrc
```

Examples

To specify the paper types for a collection of printers:

```
printer.dp1.paper:          A3
printer.ps0.paper:          letter
printer*paper:              A4
```

sets the paper type for `dp1` to `A3`, for `ps0` to `letter`, and everything else to `A4`.

Note that `$AIPSROOT/.aipsrc` should only be used to define resources which are not directly relevant to a particular AIPS++ system (as defined by the architecture in `aipshosts` (p35)). This is prompted by the desire that where there are several systems each should be self-contained and independent of the others. Examples of resources which may appropriately be defined in `$AIPSROOT/.aipsrc` would be those relating to *inhale* (p78)), since *inhale* can be thought of as operating at the `$AIPSROOT` level.

`$AIPSARCH/aipsrc` contains default values for all recognized resources and is maintained by the system. It should not be modified; overriding definitions should instead be made in any of the other `aipsrc` files in the sequence.

See also

getrc (p43), query AIPS++ resource database.
affirm (p53), get the Boolean value of a set of arguments.

Author

Original: 1992/03/05 by Mark Calabretta, ATNF.

5.4 *.cshrc*

AIPS++ `.cshrc` file.

Synopsis

```
source .cshrc
```

Description

The `.cshrc` file which resides in `$AIPSROOT` is installed there as part of the AIPS++ system to provide a minimal environment in C-like shells (*csh*, *tcsh*) for the AIPS++ manager.

If ***inhale*** (p78) is being run to keep the AIPS++ installation in step with the master then any changes made to this file would be overwritten whenever the system master was modified. Additional definitions or redefinitions may instead be made in a separate `.cshrc_local` file which is invoked by `.cshrc` as its last operation.

The commands contained in `.cshrc` are executed immediately upon login and upon creation of every C-shell. They usually consist of definitions of aliases and C-shell variables since these are *not* inherited by subprocesses, although certain important shell variables (`argv`, `cwd`, `home`, `path`, `prompt`, `shell`, `status`, `term`, and `user`) *are* automatically set.

Environment variable definitions are normally made in the `.login` file since the environment *is* automatically inherited by every subprocess. This includes `PATH` and `MANPATH` in particular. However, remote shells such as are activated by *rsh* do not read the `.login` file (assuming that *csh* is the default login shell). If these environment variables need to be defined automatically in such cases the following line should be added to the `.cshrc_local` file

```
if (! $?term) source $AIPSROOT/.login
```

Note, however, that the ***.rshexec*** (p49) command should normally be used for remote execution.

The C-shell startup files are read in the following order:

```
.cshrc
.cshrc_local
.login
.login_local
```

See also

AIPS++ variable names (§1.2, p8).
aipsinit (p31), AIPS++ initialization procedure.
`.login` (p45), *aips2mgr*'s `.login` file.
`.profile` (p47), *aips2mgr*'s `.profile` file.
`.prompt` (p48), Set the C-shell `prompt` variable.
.rshexec (p49), AIPS++ remote execution utility.

Author

Original: 1992/05/01 by Mark Calabretta, ATNF.

5.5 *.exrc*

AIPS++ `.exrc` file.

Synopsis

Default environment for ***vi***.

Description

The `.exrc` file which resides in `$AIPSROOT` is installed there as part of the AIPS++ system to initialize the ***vi*** environment for the AIPS++ manager.

If ***inhale*** (p78)) is being run to keep the AIPS++ installation in step with the master then any changes made to this file would be overwritten whenever the master version was modified.

Author

Original: 1992/05/02 by Mark Calabretta, ATNF.

5.6 *getrc*

Query the AIPS++ resource database.

Synopsis

getrc [-*i*] [-*v*] keyword

Description

getrc fetches keyword values from the `aipsrc` (p38) databases. The mechanism has some similarities with that of the `.Xdefaults` database in X-windows. *getrc* searches a set of `aipsrc` databases (listed below) in sequence to find a match for the keyword specified on the command line.

Options

-*i* Ignore the user's `/.aipsrc` file. This is useful to obtain keyword values over which the user is not allowed to have control.

-*v* Verbose mode. *getrc* reports on `stderr` the name of the file in which it matched the keyword, and also the matching entry. Primarily used for debugging.

Resources

All AIPS++ resources are within the scope of *getrc*.

Diagnostics

getrc returns 0 on success with the value on `stdout`. Otherwise, it returns a message on `stderr` with a non-zero status as follows

Status return values
0: success
1: usage error
2: `$AIPSPATH` not defined
3: `$HOME` not defined
4: invalid regular expression
5: pattern not matched

Examples

To get the default printer (Bourne shell):

```
PRI=‘getrc printer.default 2> /dev/null‘
```

To see where the default printer definition was found:

```
grus% getrc -v printer.default
sw_hutt
Line matched in file /aips++/sun4/epping/grus/aipsrc was
printer*default:                sw_hutt
```

Files

The following `aipsrc` files are consulted in sequence, except that `/.aipsrc` is ignored if the *-i* option is specified.

```
~/.aipsrc
$AIPSROOT/.aipsrc
$AIPSHOST/aipsrc
$AIPSSITE/aipsrc
$AIPSARCH/aipsrc
```

It is not an error for any of these files to be absent or empty. Refer to the entry for `aipsrc` (p38) for an explanation of how the `aipsrc` files are used.

See also

AIPS++ variable names (§1.2, p8).
`aipsrc` (p38), AIPS++ resource database.
affirm (p53), get the Boolean value of a set of arguments.

Author

Original: 1992/03/05 by Mark Calabretta, ATNF.

5.7 *.login*

AIPS++ `.login` file.

Synopsis

```
source .login
```

Description

The `.login` file which resides in `$AIPSROOT` is installed there as part of the AIPS++ system to provide a minimal interactive environment in C-like shells (*csh*, *tcsh*) for the AIPS++ manager (assuming that one of these is the default login shell).

If *inhale* (p78) is being run to keep the AIPS++ installation in step with the master then any changes made to this file would be overwritten whenever the system master was modified. Additional definitions or redefinitions may instead be made in a separate `.login_local` file which is invoked by `.login` (p45) as its last operation before invoking `aipsinit.csh` (p31).

If `.login_local` is present it must define the `PATH` and `MANPATH` environment variables, but must not invoke *aipsinit* (p31). If `.login_local` is not present `.login` will supply reasonable but probably inadequate defaults for `PATH` and `MANPATH`.

The commands contained in `.login` are executed after `.cshrc` upon login (or if the C-shell is invoked via a name beginning with "-"). They usually consist of terminal setup commands and environment variable definitions which are automatically inherited by every subprocess.

Definitions of aliases and C-shell variables are *not* inherited by subprocesses and so are usually made in `.cshrc` which is invoked by every shell (although certain important shell variables `argv`, `cwd`, `home`, `path`, `prompt`, `shell`, `status`, `term`, and `user` *are* automatically propagated to each subprocess).

Remote shells such as are activated by *rsh* do not read the `.login` file (assuming that *csh* or *tcsh* is the default shell). If the `.login` file needs to be invoked automatically in such cases the following line should be added to the `.cshrc_local` file

```
if (! $?term) source $AIPSROOT/.login
```

Note, however, that the *.rshexec* (p49) command should normally be used for remote execution.

The C-shell startup files are read in the following order:

```
.cshrc
.cshrc_local
.login
.login_local
```

See also

AIPS++ variable names (§1.2, p8).

aipsinit (p31), AIPS++ initialization procedure.
`.cshrc` (p40), *aips2mgr*'s `.cshrc` file.
`.profile` (p47), *aips2mgr*'s `.profile` file.
.rshexec (p49), AIPS++ remote execution utility.

Author

Original: 1992/05/01 by Mark Calabretta, ATNF.

5.8 *.profile*

AIPS++ `.profile` file.

Synopsis

```
. .profile
```

Description

The `.profile` file which resides in `$AIPSROOT` is installed there as part of the AIPS++ system to provide a minimal environment for Bourne-like shells (***sh***, ***ksh***, ***bash***) for the AIPS++ manager.

In particular, `.profile` is used to define the environment for the AIPS++ manager's ***cron*** jobs, and also for ***.rshexec*** (p49).

If ***inhale*** (p78)) is being run to keep the AIPS++ installation in step with the master then any changes made to this file would be overwritten whenever the system master was modified. Additional definitions or redefinitions may instead be made in a separate `.profile_local` file which is invoked by `.profile` as its last operation before invoking `aipsinit.sh` (p31).

If `.profile_local` is present it must define the `PATH` and `MANPATH` environment variables, but must not invoke ***aipsinit*** (p31). If `.profile_local` is not present `.profile` will supply reasonable but probably inadequate defaults for `PATH` and `MANPATH`.

The commands contained in `.profile` are executed upon login (or if the shell is invoked via a name beginning with "-"). They usually consist of terminal setup commands and environment variable definitions which are automatically inherited by every subprocess.

Remote shells such as are activated by ***rsh*** do not read the `.profile` file (assuming that ***sh***, ***ksh*** or ***bash*** is the default login shell). In such cases the ***.rshexec*** (p49) command should be used.

The Bourne-shell startup files are read in the following order:

```
.profile
.profile_local
```

See also

The unix manual page for ***cron***(1).
AIPS++ variable names (§1.2, p8).
aipsinit (p31), AIPS++ initialization procedure.
`.cshrc` (p40), *aips2mgr*'s `.cshrc` file.
`.login` (p45), *aips2mgr*'s `.login` file.
.rshexec (p49), AIPS++ remote execution utility.

Author

Original: 1992/04/30 by Mark Calabretta, ATNF.

5.9 *.prompt*

Set the `prompt` variable for *csh*-like shells.

Synopsis

```
alias cd 'chdir \!*; source $AIPSROOT/.prompt'
source $AIPSROOT/.prompt
```

Description

The *cd* command is aliased in `.cshrc` (p40) to invoke `.prompt` whenever the current working directory is changed (as above).

For *csh* and *tcsh* running in *xterm*, *suntool*, and *cmdtool* windows, `.prompt` encodes an escape sequence in the `prompt` shell variable which identifies the subprocess depth, the user and host names, and current working directory in the window label, and also the host name and command number in the interactive prompt.

Since *csh* in some systems does not process escape sequences encoded into the `prompt` variable (for example, AIX and Solaris 2.2), `.prompt` recognizes a shell variable `noAutoWinLabel` which causes it simply to echo the escape sequence on `stdout` and set `prompt` to print the host name and command number. However, this method fails to reset the window label on logout from a subshell.

For other windows or ordinary terminals `.prompt` simply encodes the user and host names, the last two elements of the current working directory, and command number into the `prompt` shell variable.

See also

AIPS++ variable names (§1.2, p8).
`.cshrc` (p40), *aips2mgr*'s `.cshrc` file.

Author

Original: 1992/05/01 by Mark Calabretta, ATNF.

5.10 *.rshexec*

Remote execution utility.

Synopsis

rsh remotehost `$AIPSROOT/`*.rshexec* `[-`*i*`] [-`*e* `aips_ext] command [options]`

Description

.rshexec defines the AIPS++ environment for a process invoked by *rsh* and then *exec*'s the command with any specified arguments.

Options

-*i* Report the command before *exec*'ing it.

-*e* aips_ext Specify an alternative AIPS++ architecture extension, `aips_ext` (see *aipsinit* (p31)).

Examples

To see what the `$AIPSPATH` environment variable is defined to be on remote host *lynx*:

```
rsh lynx $AIPSROOT/.rshexec echo \''$AIPSPATH'\'
```

Note here that `$AIPSROOT` is evaluated on the local machine and it is assumed that it has the same value on the remote machine. Also note that the `$AIPSPATH` argument is protected by two sets of single quotes, the inner pair prevents the environment variable from being evaluated on the local machine, and the outer pair which is escaped to protect them from the local shell protects the environment variable from being evaluated by the login shell on the remote machine.

See also

The unix manual page for *rsh*(1).
aipsinit (p31), AIPS++ initialization procedure.
`.profile` (p47), *aips2mgr*'s `.profile` file.
inhale (p78), AIPS++ code import utility.

Author

Original: 1993/07/14 by Mark Calabretta, ATNF.

Chapter 6

General

General AIPS++ utilities [1].

6.1 *adate*

AIPS++ date utility.

Synopsis

adate [-*d*] [-*l* | -*u*]

Description

adate reports the date in the standard AIPS++ format, viz

```
Wed 1992/06/03 14:59:07 GMT
```

This format avoids the confusion caused by `mm/dd/yy` and `dd/mm/yy` formats, and is sortable on the second and/or third fields. Providing four digits for the year allows for dates past the year 2000. Reporting times in Greenwich Mean Time (GMT) reflects the international nature of the AIPS++ project.

Options

-*d* Omit the day-of-week.

-*l* Report the local time instead of GMT.

-*u* Report GMT (default).

[1] Last change: Id: General.tex,v 19.0 2003/07/16 04:18:26 aips2adm Exp

Diagnostics

Status return values
- `0`: success
- `1`: invalid option

Examples

The following construct is typical of that used in the AIPS++ code distribution scripts for maintaining the `VERSION` and `LOGFILE` files.

```
echo "$NEWVERSN `adate` ($MODE)" | cat - LOGFILE_ > LOGFILE
```

See also

The unix manual page for the *date*(1) command on which *adate* is based.

Author

Original: 1992/06/03 by Mark Calabretta, ATNF.

6.2 *affirm*

Determine the Boolean value of a set of arguments.

Synopsis

affirm `[-`*a* `|` `-`*o*`]` `[-`*s*`]` `[string1 [string2...]]`

Description

affirm tests its arguments for truth or falsehood. The arguments can be of the form `true/false`, `t/f`, `yes/no`, `y/n`, `on/off`, `1/0` and are case insensitive. It will print either `true` or `false` on `stdout` with a status return to match.

Options

`-`*a* Produce the logical AND of all arguments (default).

`-`*o* Produce the logical OR of all arguments.

`-`*s* Work silently, just producing the exit status.

Notes

- The `-`*a* option returns immediately if it encounters a false argument; remaining arguments are not checked for validity. It can best be thought of as searching for a `false` argument. Even if the argument list contains an unrecognized value the search still continues for a `false` value. If none is found and there were no unrecognized arguments then it returns `true`.

  The `-`*o* option behaves similarly except that it searches for a `true` argument. Hence, to test the logical value of an environment variable which may be undefined use

  ```
  affirm -a $VAR
  ```

  (or just `affirm $VAR`) to get a true result if `VAR` is blank, and use

  ```
  affirm -o $VAR
  ```

  to get a false result if `VAR` is blank.

- The `-`*a* and `-`*o* options may be used together, for example:

  ```
  affirm -o $V1 $V2 -a $V3 $V4
  ```

  returns `true` if either of `V1` or `V2` is true, or if `V3` and `V4` are both `true` or both blank.

Diagnostics

Status return values reflect a true or false result, or invokation error

`0`: true
`1`: false
`2`: unrecognized argument

Examples

To test an *aipsrc* (p38) resource setting (Bourne shell):

```
ENABLED=`affirm -a \`getrc inhale.base.code.preserve 2> /dev/null\``
```

Sets `ENABLED` to `true` if the `inhale.base.code.preserve` was set to some variant of the affirmative, or if it was not defined. To get a `false` value by default the *-o* option could have been used instead.

See also

`aipsrc` (p38), AIPS++ resource database.
getrc (p43), query AIPS++ resource database.

Author

Original: 1993/09/01 by Mark Calabretta, ATNF

6.3 *amkdir*

Create a sequence of directories.

Synopsis

> *amkdir* [-*g* group] [-*p* permissions] [-*s*] [-*v*] [directory]

Description

amkdir creates directories giving them the specified group ownership and permissions. It will create all directories in the pathname which do not already exist. The pathname may be absolute or relative.

Options

-*g* **group** Group ownership to be applied to newly created directories via *chgrp*.

-*p* **permissions** Directory permissions to be applied to newly created directories via *chmod*. Numeric or symbolic forms are acceptable.

-*s* Strip off the last component of the pathname.

-*v* Verbose mode.

It is not an error for the directory pathname to be omitted.

Diagnostics

Status return values
- 0: success
- 1: initialization error
- 2: directory creation error

Examples

Create the `dir1` and `dir1/dir2` subdirectories under `/tmp`:

```
amkdir -g staff -p ug=rwx,o=rx,g+s -s -v /tmp/dir1/dir2/file
```

The -*s* option here strips off the last component of the pathname which is `file`. This is often useful in shell scripts and makefile rules.

See also

The unix manual page for ***chgrp***(1).
The unix manual page for ***chmod***(1).
The unix manual page for ***mkdir***(1).

Author

Original: 1994/02/08 by Mark Calabretta, ATNF

6.4 *dox*

Select and view AIPS++ documents online.

Synopsis

> *dox* [*-d* directory | *--directory*=directory] [*-D* default | *--default*=default]
> [*-r* | *--restrict-search*] [document]

Description

dox provides a convenient user interface for selecting and viewing the documents stored in a specified directory and its subdirectories, `$AIPSDOCS` by default (§1.2, p8).

If no document is specified on the command line then ***dox*** will produce a list of all available files and ask the user to pick one. Full minimum-match is applied to the user's response, including minimum-match on the subdirectory name. If a case-sensitive match fails, ***dox*** tries for a case-insensitive match.

If a document is specified on the command line then full minimum-match is applied to it (including matching on subdirectories). Likewise for any default specified.

dox recognizes DVI, HTML, PostScript, and ASCII files, and also many standard unix file types (via their file suffix). It also handles compressed PostScript and text files with `.gz`, `.z`, and `.Z` suffix extensions.

Documents produced by TeX/LaTeX with a `.dvi` suffix will be displayed by ***xdvi*** (or as specified by the `DVIVIEWER` environment variable). PostScript files with a `.ps` suffix will be displayed by ***ghostview*** (or as specified by the `PSVIEWER` environment variable). HTML files with a `.html` or `.htm` suffix will be displayed by ***netscape*** (or as specified by the `HTMLVIEWER` environment variable). Ordinary text files with a `.text` or `.txt` suffix are displayed with ***more*** (or as specified by the `PAGER` environment variable).

If ***dox*** does not recognize the file suffix it will check first to see whether the file contains PostScript, then whether it is a printable ASCII file. If all else fails, ***dox*** resorts to using ***od***.

Options

***-d* directory | *--directory*=directory** The documentation directory or directories to search. If not specified, the default is taken from the `DOXPATH` environment variable if defined, otherwise `$AIPSROOT/docs`.

***-D* default | *--default*=default** Default document when querying the user interactively. The default for the default is `reference/Introaips.ps`.

-r* | *--restrict-search Restrict the search of the documentation directory and all subdirectories to `.dvi`, `.ps*`, `.text*`, and `.txt*` files when preparing the document list. The default is to list everything.

Whitespace is allowed between short-form options and their arguments.

Notes

- The matching algorithm gives more weight to the start of the filename than the middle or end. For example `Foo` matches `FooBar` before `BarFoo`. On the other hand, `foo` matches `Barfoo` before `FooBar` because it is a better case-sensitive match. The algorithm also gives more weight to the filename than directory name. For example, `Foo` matches `Bar/Foo`, or even `Bar/FooBar`, before `Foo/Bar`.
- ***dox*** starts appreciably faster if given a case-sensitive match for the start of the document name.

Diagnostics

Status return values
  `0`: success
  `1`: initialization error
  `2`: error getting document name
  `3`: error starting previewer

Examples

View the "AIPS++ *System manual*" (i.e. this manual):

```
dox System
```

Select from all available AIPS++ documents:

```
dox
```

See also

The manual page for ***dvips***(1).
The GNU manual page for ***ghostview***(1).
The GNU manual page for ***gzip***(1).
The unix manual page for ***od***(1).
The manual page for ***xdvi***(1).
The GNU manual page for ***zcat***(1).
The GNU manual page for ***zmore***(1).

Author

Original: 1993/10/29 by Mark Calabretta, ATNF

Chapter 7

Printer

Printer utilities provided with AIPS++ [1].

7.1 *lwf_a*

ASCII to POSTSCRIPT filter.

Synopsis

lwf_a [-*1*] [-*b*] [-*c*#] [-*d*#] [-*f*font] [-*g*#] [-*i*#] [-*l*] [-*n*file] [-*o*list] [-*p*#] [-*r*] [-*s*#] [-*t*#] [-*u*] [-*x*] [file1 [file2 ...]]

Description

lwf_a is a POSTSCRIPT filter. It can process multiple input files into a single multi-document POSTSCRIPT file. If no files are specified input is read from `stdin`.

lwf_a correctly processes form feed characters and tabs and also understands backspacing. If a fixed-width font is used, such as one of those in the Courier family, then backspaces followed by underscores may be used for underlining.

lwf_a can also display 7-bit and 8-bit ASCII control codes as described by the *-d* option below.

Options

-1 Print in single-sided mode; the default is to print in two-sided (duplex) mode if the printer supports it.

-b Normally a header is written at the top of each page. The *-b* flag prevents this and instead produces a separate banner page.

[1] Last change: Id: Printer.tex,v 19.0 2003/07/16 04:18:30 aips2adm Exp

`-c#` Specifies multicolumn output. The page is divided into the specified number of equally spaced columns without regard to whether the text fits within them. The default is 1.

`-d#` Display ASCII control codes (ASCII `00-1f`, `7f-ff`):

- 0: Interpret `BS`, `HT`, `NL`, `NP`, and `CR` and ignore all other control codes.
- 1: Interpret `NUL`, `BS`, `HT`, `NL`, `NP`, and `CR` and display all other control codes in two-character mnemonic form (default).
- 2: Display all control codes in two-character mnemonic form and also interpret `HT`, `NL`, `NP`, and `CR`.
- 3: Same as 2 except that lines are broken at 80 characters. Note that since horizontal tabs are still interpreted the printed line may exceed 80 columns.
- 4: Display all control codes in two-character mnemonic form and don't interpret any of them. Lines will be broken at 80 characters.
- 5: Interpret the input as a non-ASCII byte stream, displaying each byte in hexadecimal notation. Lines will be broken at 80 characters.

Modes 1, 2, and 3 are suitable for printing text files which may contain a few control codes, for example a `.cshrc` file which sets variables containing terminal escape sequences. Modes 2 and 3 might be useful for debugging makefiles since horizontal tabs in these have a special significance. Modes 4 and 5 are suitable for dumping binary files. Mode 4 might be used for examining the ASCII character strings contined in an object module, while mode 5 might be used for dumping a data file.

The two-character mnemonics are printed at half size.

ASCII control codes are listed below.

`-ffont` Set the font type, the default is Courier. Courier, Helvetica, and Times are PostScript standards; the following fonts are supported by most PostScript printers:

```
AGB    AvantGarde-Book
AGBO   AvantGarde-BookOblique
AGD    AvantGarde-Demi
AGDO   AvantGarde-DemiOblique
BD     Bookman-Demi
BDI    Bookman-DemiItalic
BL     Bookman-Light
BLI    Bookman-LightItalic
C      Courier
CO     Courier-Oblique
CB     Courier-Bold
CBO    Courier-BoldOblique
H      Helvetica
HB     Helvetica-Bold
HO     Helvetica-Oblique
HBO    Helvetica-BoldOblique
HN     Helvetica-Narrow
HNB    Helvetica-Narrow-Bold
```

HNO	Helvetica-Narrow-Oblique
HNBO	Helvetica-Narrow-BoldOblique
NCSR	NewCenturySchlbk-Roman
NCSB	NewCenturySchlbk-Bold
NCSI	NewCenturySchlbk-Italic
NCSBI	NewCenturySchlbk-BoldItalic
PR	Palatino-Roman
PB	Palatino-Bold
PI	Palatino-Italic
PBI	Palatino-BoldItalic
TR	Times-Roman
TB	Times-Bold
TI	Times-Italic
TBI	Times-BoldItalic
ZCMI	ZapfChancery-MediumItalic

The full font name (exactly as specified above) or the abbreviation given on the left may be used.

***-g*#** Group lines of text in multiples, as specified, by printing grey bands in the background. The default is not to produce banding.

***-i*#** The distance in centimetres to indent all text from the left-hand edge of the page instead of the default 1 cm.

-l Use landscape format instead of the default portrait format. The default font type, font size, and indentation allow 80 characters in portrait, and 132 in landscape format.

***-n*`file`** Use `file` as the file name in the page header.

***-o*`M:N`** Only pages whose page numbers appear in the comma separated list of numbers and ranges will be printed. A range `M:N` means pages `M` through `N` inclusive. An initial `:N` means from the beginning to page `N`, and a final `M:` means from `M` to the end. The default, "`:`", is to print all pages.

Formfeeds are inserted to maintain the correct parity when printing in duplex mode (see the ***-x*** option).

***-p*#** Set the paper type. The default is `A4` and recognized values are

```
-----------------------------------------------------
  name         point       inch          cm
-----------------------------------------------------
 A3          (842 x 1190)             29.7  x 42.0
 A4          (595 x  842)             21.0  x 29.7
 A5          (420 x  595)             14.82 x 21.0
 B4          (729 x 1032)             25.72 x 36.41
 B5          (516 x  729)             18.20 x 25.72
 statement   (396 x  612)  5.5 x 8.5  (13.97 x 21.59)
 executive   (540 x  720)  7.5 x 10   (19.05 x 25.40)
 quarto      (610 x  780)             (21.52 x 27.52)
```

```
letter      (612 x  792)  8.5 x 11   (21.59 x 27.94)
folio       (612 x  936)  8.5 x 13   (21.59 x 33.02)
legal       (612 x 1008)  8.5 x 14   (21.59 x 35.56)
10x14       (720 x 1008)  10  x 14   (25.40 x 35.56)
tabloid     (792 x 1224)  11  x 17   (27.94 x 43.18)
ledger     (1224 x  792)  17  x 11   (43.18 x 27.94)
```

In addition, `A4/letter` is recognized as being the width of `A4` and height of `letter` size paper.

-*r* Reverse the page order and file sequence. This may be useful for printers like the Apple LaserWriter which delivers pages face up, but is not appropriate for those which deliver them face down.

-*s*# Set the font size, in points. The default is 10 and legal values lie in the range from 5 to 1000.

-*t*# Set the line spacing. The default is 1.0 (single space) and legal values lie in the range from 1.0 to 3.0 (fractional values are allowed).

-*u* Invert text on the reverse side of the page when printing in duplex mode. This may be used when the pages are to be bound on the upper edge of the paper (as defined by the orientation of the text on the front side).

-*x* Exchange parity when printing in duplex mode so that the odd numbered pages are printed on the reverse side of the paper.

Notes

- *lwf_a* produces PostScript which conforms to level 3.0 document structuring conventions.
- The default font and font size (Courier 10pt) allows 80 characters per line in portrait mode, and 132 characters in landscape on `A4` paper. However, the left margin is then too narrow to be punched for a ring binder. If this is required, Courier 9pt with a 2 cm margin may be used, specify -*i* 2 -*s* 9.
- The following table lists the number of rows and columns per page for a variety of input options. It applies only for Courier with single spacing and the default indentation.

	A3		A4		letter	
Font	Portrait	Landsc.	Portrait	Landsc.	Portrait	Landsc.
size	row/col	row/col	row/col	row/col	row/col	row/col
5	220/266	150/389	150/184	101/266	140/190	104/250
6	183/222	125/329	125/153	84/222	117/158	87/209
7	157/190	107/274	107/131	72/190	100/135	74/179
8	137/166	94/239	94/115	63/166	87/118	65/156
9	122/148	83/213	83/102	56/148	78/105	58/139
10	110/133	75/192	75/92	50/133	70/95	52/125

11	100/121	68/174	68/83	46/121	63/86	47/114
12	91/111	62/160	62/76	42/111	58/79	43/104
15	73/89	50/128	50/61	33/89	46/63	34/83
18	61/74	41/106	41/51	28/74	39/53	29/69
24	45/55	31/80	31/38	21/55	29/39	21/51

- The ASCII codes in hexadecimal, with two-character mnemonics for control codes, are as follows:

```
00  NU  (NUL - null character)
01  SH  (SOH - start of heading)
02  SX  (STX - start of text)
03  EX  (ETX - end of text)
04  ET  (EOT - end of transmission)
05  EQ  (ENQ - enquiry)
06  AK  (ACK - acknowledge)
07  BL  (BEL - bell)
08  BS  (BS  - backspace)
09  HT  (HT  - horizontal tab)
0a  NL  (NL  - new line, or LF - line feed)
0b  VT  (VT  - vertical tab)
0c  NP  (NP  - new page, or FF - form feed)
0d  CR  (CR  - carriage return)
0e  SO  (SO  - shift out)
0f  SI  (SI  - shift in)

10  DL  (DLE - data link escape)
11  D1  (DC1 - device control 1)
12  D2  (DC1 - device control 2)
13  D3  (DC1 - device control 3)
14  D4  (DC1 - device control 4)
15  NK  (NAK - negative acknowledge)
16  SY  (SYN - synchonous idle)
17  EB  (ETB - end of transmission block)
18  CN  (CAN - cancel)
19  EM  (EM  - end of medium)
1a  SB  (SUB - substitute)
1b  ES  (ESC - escape)
1c  FS  (FS  - file separator)
1d  GS  (GS  - group separator)
1e  RS  (RS  - record separator)
1f  US  (US  - unit separator)

20 < >    30  0     40  @     50  P     60  `     70  p
21  !     31  1     41  A     51  Q     61  a     71  q
22  "     32  2     42  B     52  R     62  b     72  r
23  #     33  3     43  C     53  S     63  c     73  s
24  $     34  4     44  D     54  T     64  d     74  t
```

```
25  %     35  5     45  E     55  U     65  e     75  u
26  &     36  6     46  F     56  V     66  f     76  v
27  '     37  7     47  G     57  W     67  g     77  w
28  (     38  8     48  H     58  X     68  h     78  x
29  )     39  9     49  I     59  Y     69  i     79  y
2a  *     3a  :     4a  J     5a  Z     6a  j     7a  z
2b  +     3b  ;     4b  K     5b  [     6b  k     7b  {
2c  ,     3c  <     4c  L     5c  \     6c  l     7e  ~
2d  -     3d  =     4d  M     5d  ]     7c  |     6d  m
2e  .     3e  >     4e  N     5e  ^     7d  }     6e  n
2f  /     3f  ?     4f  O     5f  _     6f  o

7f DL (DEL - delete)
```

Eight-bit codes (from `80` to `ff`) are displayed via their two hexadecimal digits using "`abcdef`" (in lower case) to distinguish them from certain seven-bit control codes all of which are printed in upper case - in particular, the eight-bit codes (`d1`, `d2`, `d3`, `d4`, and `eb`) and the seven-bit codes (`D1`, `D2`, `D3`, `D4`, and `EB`).

Do not confuse the hexadecimal code `ff` with "formfeed" which is displayed as `NP` "new-page".

Examples

The command

```
lwf_a -s12 -i2.5 file1
```

would convert `file1` to PostScript, indented 2.5 cm, and in 12pt Courier font.

The command

```
lwf_a -b -c2 -fPR -s6 foo.f
```

would convert `foo.f` to PostScript in two column form using a 6pt Palatino font. A separate banner page would be printed.

The command

```
man lwf_a | lwf_a
```

would convert the manual page for *lwf_a* to PostScript.

Bugs

If you find any bugs please report them to *mcalabre@atnf.csiro.au*.

See also

The unix manual page for ***lpr***(1).
pri (p68), print ASCII or POSTSCRIPT files.

Author

Barry Brachman wrote the original version called ***lwf***.
Extensively modified by Mark Calabretta, 1989/May - 1994.

7.2 *pra*

Print AIPS++ class implementation files.

Synopsis

pra [-*l*] [-*P* printer] [class1 [class2 ...]]

Description

pra prints checked-in AIPS++ class files in a compact form with the header file followed by the class implementation file.

Unless the -*l* option is specified, *pra* only prints checked-in code in the $AIPSROOT directories, any code in the user's own workspace is ignored.

If no AIPS++ classes are specified, all classes in the current directory will be printed.

If no printer is specified via the -*P* option, the printer is that returned by the *prd* (p67) command.

Options

-*l* Print programmer class files from the current directory if they exist.

-*P*printer Printer to use instead of the default (see *prd* (p67)).

Examples

The command

```
cd ~/aips++/code/aips/implement/Tables
pra Table
```

would print `Table.h`, and `Table.cc` in two-column landscape format.

See also

AIPS++ variable names (§1.2, p8).
lwfa (p59), PostScript filter.
prd (p67), report the default printer.
pri (p68), print ASCII or PostScript files.
prm (p70), delete entries from the printer queue.
prq (p71), list contents of the default printer queue.

Author

Original: 1992/03/04 by Mark Calabretta, ATNF.

7.3 *prd*

Report the default printer.

Synopsis

prd

Description

prd returns the name of a host's (or user's) default AIPS++ printer on `stdout`. If the `PRINTER` environment variable is defined ***prd*** returns its value, otherwise it queries the AIPS++ resource database.

Options

None.

Resources

- `printer*default`: Default printer (no default).

Examples

The command

```
prd
```

reports the default AIPS++ printer.

See also

getrc (p43), query AIPS++ resource database.
lwfa (p59), PostScript filter.
pra (p66), print AIPS++ class implementation files.
pri (p68), print ASCII or PostScript files.
prm (p70), delete entries from the printer queue.
prq (p71), list contents of the default printer queue.

Author

Original: 1992/03/04 by Mark Calabretta, ATNF

7.4 *pri*

Print ASCII or POSTSCRIPT files.

Synopsis

pri [-*m* mode] [-*p* paper] [-*P* printer] [-*t* type] [-*w*] [file1 [file2 ...]]

Description

pri prints ASCII or POSTSCRIPT files on a POSTSCRIPT printer with a variety of formatting options. If no files are specified the input is taken from `stdin`.

If the file is in standard POSTSCRIPT form in which the first line begins with `%!` it will be sent to the POSTSCRIPT printer without conversion. Otherwise, it will be converted to POSTSCRIPT using the ***lwf_a*** (p59) utility. This behaviour can be explicitly overridden with the **-*t*** option.

If no printer is specified via the **-*P*** option, the printer is that returned by the ***prd*** (p67) command.

Options

A space is allowed between the option letter and its value (if any).

-*m*`mode` Print mode for ASCII files (doesn't apply to POSTSCRIPT):

`p`	Portrait mode.
`l`	Landscape mode.
`72`	Two-column portrait mode for files with lines not exceeding 72 characters in length.
`80`	Two-column landscape mode for files with lines not exceeding 80 characters in length.

-*p*`paper` Paper type (doesn't apply to POSTSCRIPT):

`3 or A3`	Metric A3
`4 or A3`	Metric A4
`l or letter`	Letter

-*P*`printer` Printer to use instead of the default (see ***prd*** (p67)). ***pri*** recognizes a `ghostview` pseudo-printer which may be used to "print" the file to a ***ghostview*** POSTSCRIPT previewer on a workstation.

-*t*`type` Input file type.

`txt`	ASCII text file.
`ps`	POSTSCRIPT file.

-*w* Wait for the print queue to empty before submitting each job. ***pri*** checks the state of the queue once a minute, and will wait for up to one hour before submitting the job.

All other options are passed to ***lwf_a*** (p59).

Resources

- `printer*paper`: Paper type -

`A3`	metric `A3`
`A4`	metric `A4` (default)
`letter`	American letter size

Examples

The command:

```
pri -m 80 aips.h
```

would print `aips.h` in two-column landscape format.

See also

getrc (p43), query AIPS++ resource database.
lwf_a (p59), PostScript filter.
pra (p66), print AIPS++ class implementation files.
prd (p67), report the default printer.
prm (p70), delete entries from the printer queue.
prq (p71), list contents of the default printer queue.

Author

Original: 1992/03/04 by Mark Calabretta, ATNF.

7.5 *prm*

Delete entries from the printer queue.

Synopsis

prm [-*P* printer] job1 [job2 ...]

Description

prm deletes entries from a printer queue. If no printer is specified via the -*P* option, the printer is that returned by the *prd* (p67) command.

Options

-*P*printer Printer to use instead of the default.

Examples

The commands

```
prq
prm 578
```

would list the contents of the default queue and then delete job number 578.

See also

lwf_a (p59), PostScript filter.
pra (p66), print AIPS++ class implementation files.
prd (p67), report the default printer.
pri (p68), print ASCII or PostScript files.
prq (p71), list contents of the default printer queue.

Author

Original: 1992/03/04 by Mark Calabretta, ATNF.

7.6 *prq*

List contents of the default printer queue.

Synopsis

prq [-*P* printer]

Description

prq lists the contents of a printer queue. If no printer is specified via the ***-P*** option, the printer is that returned by the ***prd*** (p67) command.

Options

-P**printer** Printer to use instead of the default.

Examples

The command

```
prq
```

lists the contents of the default printer queue.

See also

lwf_a (p59), PostScript filter.
pra (p66), print AIPS++ class implementation files.
prd (p67), report the default printer.
pri (p68), print ASCII or PostScript files.
prm (p70), delete entries from the printer queue.

Author

Original: 1992/03/04 by Mark Calabretta, ATNF.

Chapter 8

Code distribution

AIPS++ source code distribution utilities [1].

8.1 *exhale*

Create AIPS++ source code updates.

Synopsis

exhale [-*e*] [-*l*]

Description

exhale generates updates for the AIPS++ code distribution system. It allows participating AIPS++ code development sites to maintain a slave RCS repository which tracks changes to the master (see ***inhale*** (p78)).

exhale runs on the master AIPS++ host, *aips2.nrao.edu*, and is responsible for producing the following update files which is deposits in `~ftp/pub/master` (see §1.1.5, p7):

- The base release of the AIPS++ RCS version files (see ***avers*** (p113)). These update files have names of the form `master-`*`mm`*`.000.tar.gz`, where *mm* is the two-digit major, or base-release version number. The base release files are produced infrequently, only by special request, and are preserved indefinitely.
- Incremental updates, each of which contains copies of all RCS version files which changed since the last time ***exhale*** ran. These have names of the form `master-`*`mm`*`.`*`nnn`*`.tar.gz`, where *nnn* is the three-digit minor, or incremental version number. They are kept for thirty days.
- The `VERSION` file is used as a timestamp file for the incremental updates. It also contains the major and minor version numbers of the most recent update and the time it was

[1] Last change: Id: CodeDstn.tex,v 19.0 2003/07/16 04:18:31 aips2adm Exp

produced. The update is skipped if no files have been modified since the last incremental update.

- A cumulative update file which contains a copy of every RCS version file which has changed since the last base release. This has a name of the form `master-`*mm.nnn*`.ALL.tar.gz`. Cumulative update files are kept for seven days. In addition, cumulative update files are kept for all versions listed in the `BLESSED` file as having passed the ***testsuite*** (p260) or having met whatever other criteria is deemed suitable.

  The base release file for the current major version is used as the timestamp for cumulative updates.

The update files are ***tar*** files which have been compressed by ***gzip*** for transmission to remote AIPS++ sites via ***ftp***.

exhale also maintains a `MANIFEST` of all files distributed in each incremental update, a `LOGFILE` of every invokation, and a `CHKSUM` file used by ***inhale*** (p78) for verification of received files.

exhale has many bookkeeping tasks to perform on the infrequent occasions when it is called upon to produce a new base release, *mm*`.000` (see ***avers*** (p113)):

- Make temporary copies of the master and archive on which to operate.
- Apply ***ax_master*** in "archive" mode (see ***ax*** (p115)) to rescind or rename archive files and directories.
- Create any new directories in the archive that appeared in the master during AIPS++ major version *ll*, where *ll* = *mm* - 1.
- Delete rescinded files corresponding to AIPS++ major version *kk* from the master, where *kk* = *mm* - 2. These will have been archived when the base release for version *ll* was produced.
- Perform the following operations on every RCS version file in the master repository:
  - "Outdate" RCS revisions corresponding to AIPS++ major version *kk* using the ***rcs -o*** command, and also delete any symbolic names corresponding to the outdated revisions (these older revisions will have been archived when the base release for version *ll* was produced). The old revisions are deleted in order to reduce the size of the RCS version file for efficient checkin (***ai*** (p93)) and checkout (***ao*** (p97)). This operation is only performed when ***exhale*** is invoked with the ***-e*** option.
  - "Uprev" the RCS version file to RCS revision number *m*`.0`, subsequent revisions are numbered from that point. In this process, any exant locks are carried forward from RCS revision number *l*`.`*?* to *m*`.0`. Revision *m*`.0`, which contains no differences from the previous revision, is forcibly created using the ***ci -f*** command.
  - Concatenate RCS revisions numbered *l*`.0` to *m*`.0` in the master into the archive RCS version file using a specially written utility called ***rcscat*** (p139). This includes rescinded files. Files created in the master since the previous base release are simply copied to the archive.

- Verify the master and archive using *xrcs* (p146).
- Create the base release tar file for consortium installations in `/pub/master`.
- Create the base release tar files for end-user installations in `/pub/code`.

Options

-e Expire mode; "outdates" old RCS revisions. Only useful when *exhale* is creating a new base release; ignored otherwise.

-l Latchkey mode (see notes); the update produced will be invisible to *inhale* (p78) unless it is also invoked with the *-l* option.

Resources

exhale always ignores `$HOME/.aipsrc` (see *getrc* (p43)).

- `account.manager`: Account name (and group) of the owner of the AIPS++ source code. *exhale* only allows itself to be run from this account.
- `account.programmer`: (Account name and) name of the AIPS++programmer group.

Notes

- Latchkey updates are used to restrict the AIPS++ distribution temporarily, for example, when the AIPS++ sources are undergoing significant structural changes which may take several days to complete. Instead of switching *exhale* off during this period, it may continue to be run in latchkey mode. The latchkey mechanism hides updates from general view, but slave installations which do want to track the changes may do so by specifying the *-l* latchkey option to *inhale* (p78). Latchkey mode is terminated as soon as *exhale* is invoked without the *-l* option, even if there are no files to be updated.

  The latchkey mechanism works by recording a version and timestamp on the second line of the `VERSION` file. Normally *inhale* (p78) obtains the current version number from the first line, but if invoked with the *-l* option it reads the last line instead.

- *exhale* should only to be run from the machine which holds the master copy of AIPS++ source code in Socorro.
- *exhale* disables checkins (*ai* (p93)), checkouts (*ao* (p97)), header changes (*am* (p103)), deletions (*ax* (p115)), and renames (*amv* (p105)) for the duration of its execution.
- A new base release is signalled by manually incrementing the major version number in `~ftp/pub/master/VERSION` and setting the minor version number to "-".
- *exhale* regenerates the *av* (p111) filename cache, consolidating all addenda into a single cache file.

Diagnostics

Status return values correspond to the various phases of the operation
`0`: success
`1`: initialization error
`2`: error preparing a new base release
`3`: error creating incremental or base update
`4`: error creating cumulative update
`5`: error creating public release
`6`: installation error

Examples

The following *aips2mgr* ***cron*** entry runs ***exhale*** at 0630 and 1830 each night in Socorro.

```
30 06,18 * * * (. $HOME/.profile ; exhale)
```

Bugs

- The `$AIPSMSTR` and `/pub` directory hierarchies must reside in the same unix filesystem.
- Newly created directories are not propagated if they don't contain newly created files.
- ***exhale*** does most of its work in temporary directories beneath `/pub/master` and `/pub/code`, and cleans up on receiving a catchable interrupt. However, sending it a `SIGKILL` signal (e.g. via ***kill -9***) would at the least cause it not to clean up its work directories, but if given during the brief though crucial installation phase, could cause serious damage to the integrity of the code distribution system.

See also

The unix manual page for ***cron***(1).
The unix manual page for ***ftp***(1).
The unix manual page for ***tar***(1).
The GNU manual page for ***gzip***(1).
Section §1.4, p9, AIPS++ accounts and groups.
AIPS++ variable names (§1.2, p8).
avers (p113), AIPS++ version report utility.
ax (p115), AIPS++ code deletion utility.
getrc (p43), query AIPS++ resource database.
inhale (p78), AIPS++ code import utility.
av (p111), AIPS++ filename validation utility.
rcscat (p139), concatenate two RCS version files.
testsuite (p260), maintain the most recent good cumulative update.
xrcs (p146), verify the internal consistency of RCS version files.

Author

Original: 1992/04/06 by Mark Calabretta, ATNF.

8.2 *inhale*

Retrieve AIPS++ source code updates.

Synopsis

inhale [-*b*] [-*c*/-*m*] [-*D*] [-*d*] [-*l*] [-*n*] [-*P*] [-*R* NAME] [-*r*#] [-*w*#] [-*cvsup* data|docs|all] [*binary*]

Description

inhale is a part of the AIPS++ code distribution system. It allows participating AIPS++ code development sites to maintain a slave copy of the master RCS repository which is no more than 12 hours out-of-date (see ***exhale*** (p73)).

inhale fetches the compressed ***tar*** files needed to update a consortium site's copy of the AIPS++ RCS repositories from the `/pub/master` area on the master AIPS++ host *aips2.nrao.edu*. After installing the updates, ***inhale*** invokes ***sneeze*** (p85) to rebuild the system and documentation. The rebuild may be done serially or in parallel for any number of AIPS++architectures on various hosts.

The update files are of three types (see ***exhale*** (p73)):

- A base release which contains all master RCS files at the time of the last base release.
- Incremental updates which consist of all RCS files changed since the preceding incremental update (usually two per day).
- A single cumulative update file which consists of all RCS files changed since the last base release.

inhale compares version information stored in the `/pub/master/VERSION` file on the master machine with that in `$AIPSLAVE/VERSION` on the local machine to determine which files need to be fetched.

If the `$AIPSLAVE/VERSION` file does not exist, or if the major version number stored therein is less than that of `/pub/master/VERSION`, ***inhale*** fetches the latest base release. If the `-c` option was specified, ***inhale*** will fetch the cumulative update file. Otherwise, as many of the incremental update files will be fetched as are needed to bring the system up-to-date. If there are insufficient of these stored in `/pub/master`, then ***inhale*** resorts to a cumulative update.

If an update contains the RCS version file for ***ax_master*** then it will be applied to the slave and code areas before the update is installed. This may happen several times for a multi-increment update. Cumulative updates always contain ***ax_master*** which ***inhale*** applies in cumulative mode.

If the major version number recorded in `/pub/master/VERSION` is greater than that recorded in `$AIPSLAVE/VERSION`, or if `$AIPSLAVE/VERSION` does not exist, then ***inhale*** does a "base installation". In this mode it fetches only the base release file and before installing it deletes any existing slave and code directories. After installing it, and if not contra-indicated by the `inhale.base.slave.preserve` and `inhale.base.slave.preserve` resources, *inhale* preserves a copy of the slave and code directories in `$AIPSROOT/base-MM/...`, where `MM` is the major version number of the new base release. *inhale* then invokes *sneeze* (p85) in base mode.

After doing a base installation, *inhale* reinvokes itself if necessary to bring the minor version number up-to-date.

inhale maintains the `LOGFILE` files in `$AIPSLAVE` and `$AIPSCODE`. These record the version, time of completion, and mode of the last slave and code updates (see *avers* (p113)).

Options

-b Use the most recently "blessed" release (see notes). Useful for sites that wish to track development but that also require stable releases.

-c Force a cumulative update. This should be done regularly (usually once a week) to guarantee that the slave copy does not deviate from the master.

-cvsup **data—docs—all** Update the data, docs or both using cvsup. Cvsup must be in the PATH for this option to work. If a supfile exists it will be used otherwise the supfile will be created.

-D Fetch the latest update which has passed the *testsuite* (p260).

-d This option is operative only if the *-n* option is also specified, it causes `docsys` to be omitted from the list of targets (see *makefiles* (p162). Note that even if documentation compilation is defeated via the `DOCSYS` variable in `makedefs` (p155) (see *makedefs* (p155)) this option may still be useful in preventing `docsys` from being recorded in the `LOGFILE` entry for the rebuild.

-l Latchkey mode; fetch latchkey updates (if any) together with updates intended for general distribution (see *exhale* (p73)).

-m Perform a "mixed" update: retrieve the incremental update files, then invoke *sneeze* (p85) in cumulative mode. This is useful for minimizing file transfer times when performing cumulative style updates over slow network links.

-n Don't remake the system (`allsys`) after installing the update. It does however remake the `install` target, and also `docsys` provided that the *-d* option is not also specified.

-P Use passive mode when doing FTP. Useful for firewalls etc. The FTP client being used must support passive mode.

-R **NAME** instead of downloading the default distribution, get the one called "NAME".

-r **count** Set the retry count for the ftp transfer, default is 12. Set this to zero to circumvent the *ftp*, *inhale* will then assume that all necessary files have been fetched from `/pub/master` into directory `$AIPSLAVE/tmp`, including `/pub/master/VERSION`.

-w **interval** Set the retry interval for the ftp transfer in seconds, default is 900.

binary This option will download the latest stable linux binary executables, libraries and scripts. /aips++/stable must exist (typically as a symbolic link) for the binaries to work.

Resources

inhale always ignores `$HOME/.aipsrc` (see *getrc* (p43)).

General resources.

- `account.manager`: Account name (and group) of the owner of the AIPS++ source code. *inhale* only allows itself to be run from this account.
- `account.programmer`: (Account name and) name of the AIPS++programmer group.

The following resources used by *inhale* should be defined in `$AIPSROOT/.aipsrc`.

- `inhale.base.slave.preserve`: If not false (see *affirm* (p53)) then preserve the slave repository when a new base release is installed.
- `inhale.base.code.preserve`: If not false then preserve the code areas when a new base release is installed.
- `inhale.sneeze.hosts`: List of hosts on which to invoke *sneeze* (p85) to rebuild AIPS++ for architectures which differ from that of the host on which *inhale* is executing. An architecture extension (see (*aipsinit* (p31)) may be specified by appending it to the hostname with an intervening colon, i.e. <hostname>:<aips_ext>. The `aips_ext` is explicitly permitted to be null or set to "_" to signal the host's default architecture. The *inhale* host's name will be ignored if included in the list (unless an extension is specified).
- `inhale.sneeze.$HOST.rcmd`: The command *inhale* should use to invoke *sneeze* (p85) on remote systems. Defaults to *rsh*. *ssh*, the "secure shell," is one possible alternative.
- `sneeze.sleep`: Use this option to adjust the delay time (in seconds) for asyncronous builds. The default is 600s which should be adequate to avoid the glish concurrency configure problem.

Notes

- *inhale* fetches a `CHKSUM` file which it uses to verify the integrity of the files it has received. The checksums are computed with the BSD version of *sum* which algorithm differs from that of SysV. *inhale* may complain about "files not received in good order" if *aips2mgr* uses the SysV version of *sum*, leading to repeated attempts to fetch the update files. See §2.3, p15 for advice on sidestepping this problem.
- *inhale* fetches the file `/pub/master/BLESSED` which it uses, when invoked with the *-b* option, in conjunction with the `VERSION` files to determine which files need to be fetched. *inhale* will abort if it is invoked with the *-b* option and it cannot parse the `BLESSED` file to determine the most recently-blessed version for the local system's architecture. *inhale* will default to normal behavior if the `BLESSED` file is not present. Versions are listed, individually by architecture, in `BLESSED` when they have either passed the *testsuite* (p260) or have met whatever other criteria is deemed suitable.

- If "blessed" mode is used on an ***inhale*** host that invokes ***sneeze*** (p85) on multiple client machines of different architectures then those clients will be updated to the most recently-blessed version for the ***inhale*** host's architecture, not their own.
- The incremental (and cumulative) update files contain the `VERSION` file for the *preceeding* version. This is used in multi-increment updates.
- ***inhale*** checks the directory ownerships and permissions on all AIPS++ slave and code areas, reports incorrect ownerships, and silently fixes the group ownerships and permissions where possible.
- Since ***inhale*** may invoke ***sneeze*** (p85) to rebuild the AIPS++ system which includes the ***inhale*** executable it must take precautions to avoid overwriting itself. It does so by copying itself to ***inhale-*** and then ***exec***'ing this.
- ***inhale*** is a critical part of the AIPS++ code distribution system. If a broken version was distributed to consortium sites, then it probably would not be able to fetch and install a repaired version subsequently distributed. The code distribution system would be broken.

Diagnostics

Status return values correspond to the various phases of the operation
`0`: success
`1`: initialization error
`2`: initial ftp error
`3`: main ftp error
`4`: error extracting or installing the new RCS files
`5`: error checking out the new code
`6`: error rebuilding the system

Examples

The following *aips2mgr* ***cron*** job entry runs ***inhale*** at 2300 each night performing a cumulative update on Saturday evening, and incremental updates on all other evenings:

```
# Update the AIPS++ slave directories, incremental Sun-Fri, cumulative Sat.
00 23 * * 0-5 (. $HOME/.profile ; inhale) 2>&1 | \
   mail -s "Epping inhale" aips2-inhale@nrao.edu
00 23 * * 6   (. $HOME/.profile ; inhale -c) 2>&1 | \
   mail -s "Epping inhale -c" aips2-inhale@nrao.edu
```

(Note that all ***cron*** entries must be one-liners but they are broken here for clarity.) The ***inhale*** log is emailed to the `aips2-inhale` exploder which allows the code distribution system to be monitored. An alternative method is to forward all of *aips2mgr*'s mail via the `$AIPSROOT/.forward` file, or as a mail alias.

See also

The unix manual page for *tar*(1).
The unix manual page for *cron*(1).
The unix manual page for *rsh*(1).
The unix manual page for *ssh*(1).
AIPS++ variable names (§1.2, p8).
Section §1.4, p9, AIPS++ accounts and groups.
affirm (p53), get the Boolean value of a set of arguments.
avers (p113), AIPS++ version report utility.
ax (p115), AIPS++ code deletion utility.
exhale (p73), AIPS++ code export utility.
getrc (p43), query AIPS++ resource database.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.
sneeze (p85), AIPS++ system rebuild utility.
testsuite (p260), maintain the most recent good cumulative update.

Author

Original: 1992/04/08 by Mark Calabretta, ATNF.

8.3 *inhold*

Wait for ***inhale*** (p78) to finish.

Synopsis

> ***inhold*** [-*t*#]

Description

inhold waits for ***inhale*** (p78)) to finish rebuilding the system. It may be used to synchronize a programmer rebuild, especially as initiated via a ***cron*** job, so that it starts shortly after ***inhale*** (p78) finishes.

inhold uses ***tract*** (p144) to determine when `$AIPSARCH/LOGFILE` (§1.2, p8) is updated.

Options

-*t* interval Time resolution, in seconds, to test for the completion of ***inhale*** (p78) (default 300).

Resources

none

Notes

- The time resolution specified by the -*t* option determines the sleep interval. An extra 60 seconds is added to this to derive the maximum age of the `LOGFILE`. This accounts for slow NFS updates of `LOGFILE`'s timestamp.

Diagnostics

Status return values:
`0`: success
`1`: initialization error

Examples

An example programmer ***cron*** entry might be as follows:

```
00 23 * * 6 . $HOME/.profile ; inhold && rebuild
```

where ***rebuild*** is some script written by the programmer to rebuild the AIPS++ workspace. The ***cron*** start time should coincide with the start time of ***inhale*** (p78).

See also

The unix manual page for ***cron***(1).
AIPS++ variable names (§1.2, p8).
inhale (p78), AIPS++ code import utility.
tract (p144), Report the age of a file or directory.

8.4 *sneeze*

AIPS++ system rebuild utility.

Synopsis

```
sneeze [-e extlist] [-l] [-m mode] [-s seconds] targets
```

Description

sneeze is a part of the AIPS++ code distribution system. It is invoked by ***inhale*** (p78)) to rebuild the AIPS++ system areas including object libraries, executables, and documentation. It can rebuild serially the systems for several `aips_exts` (see ***aipsinit*** (p31)).

Before rebuilding the system, ***sneeze*** invokes ***ax_master*** with the ***-system*** option to delete unwanted files and directories from the system areas. On cumulative updates it invokes ***ax_master*** in cumulative mode and this deletes all static object libraries and class template repositories in addition to applying all deletions recorded in ***ax_master***'s action list. ***sneeze*** also invokes '`gmake -C $AIPSCODE cleansys`' (§1.2, p8) on cumulative updates to further clean up the system areas.

At the end of a cumulative update ***sneeze*** invokes '`gmake -f $AIPSARCH/makedefs diagnostics`' to report the installed versions of third-party utilities essential for AIPS++ such as ***gmake*** (p154), and also to print the current values of all `makedefs` (p155) variables.

For a base installation (*-m* `base`), after the rebuild is complete and if not contra-indicated by the `sneeze.base.$ARCH.preserve` resource, ***sneeze*** preserves a copy of the system in `$AIPSROOT/base-MM/...`, where `MM` is the major version number of the new base release, and then invokes '`gmake -C $AIPSROOT/base-$NEWMAJOR/code/install aipsroot`' to remake the root level directory.

sneeze maintains the `LOGFILE` file in `$AIPSARCH` which records the version, time of completion, and ***gmake*** (p154) targets of the last rebuild (see ***avers*** (p113)).

Options

-e* extlist** Serially rebuild the systems corresponding to each `aips_ext` in the blank-, or colon-separated list. If no *-e* option is given ***sneeze just rebuilds the host's default architecture as specified in `aipshosts` (p35). An "_" in the extlist signals no extension.

If there are any more `aips_exts` to be built after ***sneeze*** has finished the current one then it reinvokes itself with the *-e* option specifying an extlist from which the current `aips_ext` has been removed.

-l Log mode - if specified then the output of sneeze will be logged to a file specified by the `sneeze.$HOST.$AIPSEXT.logfile` resource and mailed to the addresses specified by the `sneeze.$HOST.$AIPSEXT.logmail` resource. If these are not defined the log is sent to `$AIPSARCH/sneeze.log` and the mail to *aips2-inhale@nrao.edu*.

***-m* mode** Mode of the rebuild, `incremental` (default), `cumulative`, or `base`.

-s seconds Interval in seconds to sleep before commencing build. This option is used to prevent concurrent builds from configuring glish at the same time.

If no targets are specified (`makefiles` (p162)) *sneeze* remakes `allsys`.

Resources

sneeze always ignores `$HOME/.aipsrc` (see *getrc* (p43)).

General resources.

- `account.manager`: Account name (and group) of the owner of the AIPS++ source code. *sneeze* only allows itself to be run from this account.
- `account.programmer`: (Account name and) name of the AIPS++programmer group.

The following resources used by *sneeze* should be defined in `$AIPSROOT/.aipsrc`.

- `sneeze.$HOST.$AIPSEXT.logfile`: File in which the log produced by *sneeze* is preserved.
- `sneeze.$HOST.$AIPSEXT.logmail`: Addresses to which the log produced by *sneeze* is mailed.
- `sneeze.$HOST.$AIPSEXT.mailer`: Mail delivery program to use for log mailings. Defaults to *mail*.
- `sneeze.base.$ARCH.preserve`: If not false (see §6.2, p53) then preserve the system for `$AIPSARCH` in `$AIPSROOT/base-MM/...` when a new base release is installed, where `MM` is the major version number of the new base release.

See also the `inhale.sneeze.hosts` resource used by *inhale* (p78) to control how *sneeze* is invoked.

Notes

- *sneeze* is mainly intended for the use of *inhale* (p78) rather than for general usage.
- It is assumed that the code areas have already been brought up-to-date with respect to the slave repositories (by *inhale* (p78)).
- *sneeze* checks the directory ownerships and permissions on all AIPS++ system areas, reports incorrect ownerships, and silently fixes the group ownerships and permissions where possible.
- Since *sneeze* rebuilds the AIPS++ system which includes the *sneeze* executable it must take precautions to avoid overwriting itself. It does so by copying itself to *sneeze-* which it then *exec*s. This also provides an opportunity for it to redirect its own output.

Diagnostics

Status return values
  `0`: success
  `1`: initialization error

Examples

sneeze is used by ***inhale*** (p78) and should not normally be invoked manually. ***inhale*** (p78) invokes it asynchronously on remote hosts with the ***-l*** option using ***.rshexec*** (p49)), then executes it synchronously on the local host without the ***-l*** option.

See also

The unix manual page for ***mail***(1).
AIPS++ variable names (§1.2, p8).
affirm (p53), get the Boolean value of a set of arguments.
aipshosts (p35), AIPS++ hosts database.
avers (p113), AIPS++ version report utility.
ax (p115), AIPS++ code deletion utility.
getrc (p43), query AIPS++ resource database.
inhale (p78), AIPS++ code import utility.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

Author

Original: 1993/09/13 by Mark Calabretta, ATNF.

Chapter 9

Code management

AIPS++ source code management utilities [1].

9.1 *ac*

Add an entry to an AIPS++ changelog file.

Synopsis

ac [options for *ac*]

Description

ac adds an entry to an AIPS++ changelog file.

ac assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). It determines the appropriate changelog file from the current working directory.

ac invokes *ao* (p97) to check out the appropriate changelog file. It also checks out another file which is used to generate a unique id for each change. Thereafter a glish script is started to let the user fill in a form for the change and to add the change to the changelog file. The user has the opportunity to abort the operation, in which case the checked out files are unlocked. In the change is added, *ai* (p93) is invoked to check in the changed files.

Options

-i Do only a check in of the changed files. This should only be used if the normal (automatic) check in failed because an *exhale* (p73) has started or due to another error.

[1] Most recent change: Id: CodeMgmt.tex,v 19.0 2003/07/16 04:18:28 aips2adm Exp

Notes

- The AIPS++ changelog files are meant to generate an overview (by ***buildchangelog*** (p120)) of the new and changed features in an AIPS++ release. It means they are higher level than the RCS changelogs which are programmer oriented.
- Almost every source directory has a changelog file to keep track of the changes in the AIPS++ system.
  All glish subdirectories share a single changelog file in their root directory. The same is true for the system and the documentation subdirectories.
- The DISPLAY variable needs to be set, because ***ac*** uses a GUI to let the user fill in the change form.
- If checking out fails because the file does not exist yet, one should create an empty file changelog and check it in using ***ai*** (p93). In general this should only occur in new directories.
- Checking in should normally succeed, but might fail because the an ***exhale*** (p73) has started or because network connection is down. In that case ***ac*** should be run at a later time with the -i flag (to only do a checkin of the files).

Diagnostics

Status return values
`0`: success
`1`: initialization error
`2`: checkout failure
`3`: checkin failure
`4`: no change entry created by user

Examples

The command

```
cd ~/aips++/code/aips/implement/Tables
ac
```

will let the user fill in the change form and add the completed form to the changelog file of library module Tables.

Bugs

- ***ac*** does not detect why a checkout or checkin fails.

See also

rac (p129) AIPS++ ac for remote users.
ai (p93) AIPS++ code checkin utility.
ao (p97) AIPS++ code checkout utility.
buildchangelog (p120) process AIPS++ changelog files.

Author

Original: 2000/06/05 by Ger van Diepen, NFRA

9.2 *ae*

Extricate an AIPS++ source file from a half-completed checkin/out.

Synopsis

```
ae file
```

Description

ae extricates a source file in the current directory from a half-completed RCS checkin/out. Such problems sometimes arise when the checkin/out is interrupted part way through leaving behind an RCS semaphore file.

The presence of a defunct RCS semaphore file is indicated by repeated "in use" errors when a checkin or checkout of a file is attempted.

ae assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). It determines the corresponding master directory from the current working directory.

The filename argument must be a simple filename without path or RCS `,v` suffix.

ae invokes `$MSTRETCD/ae_master`, where `$AIPSMSTR` is expected to be NFS mounted from the AIPS++ master host (see §1.1.4, p5). It waits till the semaphore file is at least 15 minutes old before deleting it in case a slow RCS transaction is in progress.

Options

None.

Notes

- The semaphore file has a name of the form `,file,` and resides in the same directory as the master RCS version file (`file,v`).
- The utility which generates updates for the AIPS++ code distribution system *exhale* (p73) routinely searches for and deletes all semaphore files which are more than a day old. *ae* allows particular semaphore files to be deleted on a much shorter timescale.
- *ae* uses an AIPS++ utility called *tract* (p144) to determine the age of the RCS semaphore file.

Diagnostics

Status return values
`0`: success
`1`: initialization error

2: semaphore file not found
3: failed to delete semaphore file

Examples

The command

```
cd ~/aips++/code/aips/implement/FooBar
ae Foo.cc
```

would delete the RCS semaphore file for `Foo.cc` if it existed.

Bugs

- *ae* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The manual page for *ci* (1), the RCS checkin command.
AIPS++ variable names (§1.2, p8).
AIPS++ code management configuration (§1.1.4, p5).
ai (p93) AIPS++ code checkin utility.
ao (p97) AIPS++ code checkout utility.
mktree (p126) create AIPS++ directory hierarchy.
tract (p144) report the age of a file or directory.

Author

Original: 1995/02/23 by Mark Calabretta, ATNF.

9.3 *ai*

AIPS++ source code checkin utility.

Synopsis

ai [-*ae*] [-*af*] [options for *ci*] file1 [file2,...]

Description

ai checks in source files from the current directory into the AIPS++ master RCS repository. It is a front-end for the RCS *ci* command.

ai assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). The filename arguments specified to *ai* must be simple filenames without path or RCS `,v` suffix. *ai* determines the corresponding master directory from the current working directory.

Directories may also be created on the master by "checking them in". In this case any options for *ci* are ignored. `RCS` subdirectories are explicitly excluded from directory checkin.

If an AIPS++ copyright notice exists in the file to be checked in *ai* will ensure that it is kept up-to-date for the current year (unless the *-oc* option is specified, see below).

If the *-au* option is specified, then after checking the code into the master, *ai* copies the RCS version file back to the local slave RCS repository (if it exists), giving it a file ownership defined by the owner of *ai* (*aips2mgr* at the local site), which must itself be setuid. If this completed successfully, *ai* then checks out a plain-text copy of the code from the slave RCS repository into the corresponding code directory.

ai updates the master AIPS++ RCS repository by executing `$MSTRETCD/ai_master`, where `$AIPSMSTR` is expected to be NFS mounted from the AIPS++ master host (see §1.1.4, p5). Programmers who have an account on the AIPS++ master host, *aips2.nrao.edu*, may instead use a related utility, *rai* (p130), which invokes *ai* remotely on the master host via *ssh*.

Options

-ae For each file to be checked in, invoke *ae* (p92) to delete any pre-existing RCS semaphore file.

-af *ai* normally verifies that a file to be checked in does exist and will skip it if it doesn't on the assumption that the filename may have been misspelt. This behaviour may be overridden via the *-af* option which forces the creation of an empty RCS version file on the master. It is mainly for use by the AIPS++ "code cops" to allow the true author of a file to check in the initial version.

-au Update the slave repository after the checkin as explained above.

-oc Override the automatic update of the copyright notice. Automatic copyright updating can also be defeated *en masse* by creating an empty file called "`.oc`" in the master directory.

-I | *-l* | *-r* | *-u* | *-w*`login` These options (without revision number) are passed directly to *ci*, check the manual page for *ci* for an explanation. All other *ci* options are defeated.

Notes

- Checkins by *aips2mgr* are explicitly defeated to discourage anonymous changes.
- Code checked into the master repository will have the file ownership defined by the owner of `$MSTRETCD/ai_master` (*aips2adm*). This script must have the `setuid` bit set (see *chmod*).
- While *mktree* (p126) creates RCS symbolic links in the programmer directory hierarchy, these are not used by *ai*. Their main function is to allow *gmake* (p154) to automatically check out the `makefile` for a directory. They also allow RCS utilities such as *rlog* to

be applied readily to the slave RCS repository. ***ai*** does not make use of these RCS symbolic links but deals directly with the AIPS++ master repository by specifying absolute pathnames.

- Checkins may be disabled by creating a file `$MSTRETCD/ai_disable`. If this file contains any text it will be printed as an informative message.
- Checkin of new files in a particular directory can be restricted by creating a `.ai_control` file in the master directory. Only account names listed within that file will be permitted to check new files into that directory.
- ***ai*** uses ***av*** (p111) to verify the filename when a file is created and creates an addendum to the ***av*** filename cache for each new file checked in.
- ***ai*** maintains RCS revision numbers and symbolic names. It sets the release number on the RCS `trunk` to coincide with the major AIPS++ version number. Further, whenever a new revision of a file is checked in ***ai*** assigns it a symbolic name which matches the AIPS++ version number of the master at the time of the checkin. Assignment of these symbolic names allows the version of a file current for any particular AIPS++ version to be recovered easily.
- Files to be checked in must have read permission for user and group. ***ai*** automatically adds this.

Diagnostics

Status return values
`0`: success
`1`: initialization error
`2`: checkin failure
`3`: checked in ok to master, but slave not updated
`4`: code not updated
`5`: directory creation failed

Examples

The command

```
ai -au Foo.cc
```

would check in `Foo.cc` from the current directory into the corresponding AIPS++ master repository, updating the slave and plain-text copies in the process. The copy of `Foo.cc` in the current directory would be deleted. The command

```
ai -u *.c
```

would check in all `.c` files in the current directory and leave read-only copies behind.

Bugs

- If the network fails part-way through a checkin, or if the checkout is aborted before completion, an RCS semaphore file with a name of the form `,*,` may be left behind in the master RCS repository. The presence of this empty file will cause subsequent checkins or checkouts to report an "in use" error. If the not deleted manually by using the *ae* (p92) command, or *ai -ae*, the semaphore file will eventually be cleaned out by *exhale* (p73) which deletes all such files which are more than one day old.
- Since *ai* operates over the internet the file transfers are subject to network corruption. On a several occasions, usually when the network response was unusually poor, truncation of the last few characters of a file has been observed. A utility, *xrcs* (p146), is available to verify the internal consistency of RCS version files.
- *ai* invokes RCS commands directly on the master RCS repository, which may be NFS mounted from a machine on the other side of a slow internet link. A checkin involves reading the RCS version file from the master, inserting the new revision, then writing it back. These file transactions are handled more efficiently by *rai* (p130).
- *ai* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The manual page for *ci*(1), the RCS checkin command.
The unix manual page for *chmod*(1).
AIPS++ variable names (§1.2, p8).
AIPS++ code management configuration (§1.1.4, p5).
ae (p92), AIPS++ RCS semaphore file deletion utility.
ao (p97), AIPS++ code checkout utility.
am (p103), AIPS++ utility to change descriptive text.
amv (p105), AIPS++ code rename utility.
asme (p108), execute a command with the effective uid set to the real uid.
au (p109), AIPS++ code update utility.
av (p111), AIPS++ filename validation utility.
ax (p115), AIPS++ code deletion utility.
mktree (p126), create AIPS++ directory hierarchy.
rai (p130), invoke *ai* remotely via *ssh*(1).
xrcs (p146), verify the internal consistency of RCS version files.

Author

Original: 1992/04/28 by Mark Calabretta, ATNF.

9.4 *ao*

AIPS++ source code checkout utility.

Synopsis

ao [options for *co*] `file1 [file2 ...]`

Description

ao checks out source files from the AIPS++ master RCS repository into the current directory. It is a front-end for the RCS *co* command.

ao assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). The filename arguments specified to *ao* must be simple filenames without path except that an initial `RCS/` will be explicitly stripped off, as will any RCS `,v` suffix. *ao* determines the corresponding master directory from the current working directory.

ao checks out the source file by executing `$MSTRETCD/ao_master`, where `$AIPSMSTR` is expected to be NFS mounted from the AIPS++ master host (see §1.1.4, p5). Programmers who have an account on the AIPS++ master host, *aips2.nrao.edu*, may instead use a related utility, *rao* (p133), which invokes *ao* remotely on the master host via *ssh*.

Options

ao does not itself interpret any options but does recognize several which it passes to *co*.

-I | *-l* | *-r#* | *-u#* These options are passed directly to *co*, check the manual page for *co* for an explanation. Note that no version number may be specified for the *-l* option, and that a *-r* option with revision number may not be used together with *-l*. All other *co* options are defeated.

Note that the *-u* option may be used to unlock a checked out file without replacement.

Notes

- The RCS *co* command must have write permission on the version file in order to check it out of the master repository. The file ownership is defined by the owner of `$MSTRETCD/ao_master` (*aips2adm*). This script must have the `setuid` bit set (see *chmod*).
- While *mktree* (p126) creates RCS symbolic links in the programmer directory hierarchy, these are not used by *ao*. Their main function is to allow *gmake* (p154) to check out the `makefile` for a directory automatically. They also allow RCS utilities such as *alog* (p100) and *rlog* to be applied readily to the slave RCS repository. *ao* does not make use of these RCS symbolic links but deals directly with the AIPS++ master repository by specifying absolute pathnames.

- Checkouts may be disabled by creating a file `$MSTRETCD/ai_disable`. If this file contains any text it will be printed as an informative message.

Diagnostics

Status return values
 0: success
 1: initialization error, no checkout
 2: checkout failure

Examples

The command

```
ao -l Foo.cc
```

would check out `Foo.cc` with an exclusive lock. Then

```
ao -u Foo.cc
```

would remove the lock without checking the file in or deleting the working copy. The command

```
ao -l RCS/*,v
```

would check out everything in the directory (that is, assuming that `RCS` is a symlink into the slave directory, all files currently in the slave directory will be checked out of the master).

Bugs

- If the network fails part-way through a checkout, or if the checkout is aborted before completion, an RCS semaphore file with a name of the form `,*,` may be left behind in the master RCS repository. The presence of this empty file will cause subsequent checkins or checkouts to report an "in use" error. If the not deleted manually by using the ***ae*** (p92) command, or ***ai -ae***, the semaphore file will eventually be cleaned out by ***exhale*** (p73) which deletes all such files which are more than one day old.
- Since ***ao*** operates over the internet the file transfers are subject to network corruption. On a several occasions, usually when the network response was unusually poor, truncation of the last few characters of a file has been observed. A utility, ***xrcs*** (p146), is available to verify the internal consistency of RCS version files.
- ***ao*** invokes RCS commands directly on the master RCS repository, which may be NFS mounted from a machine on the other side of a slow internet link. A checkout involves reading the RCS version file from the master, inserting the locker's name, then writing it back. These file transfers are handled more efficiently by ***rao*** (p133).
- ***ao*** requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The manual page for *co*(1), the RCS checkout command.
The unix manual page for *chmod*(1).
AIPS++ variable names (§1.2, p8).
ae (p92), AIPS++ RCS semaphore file deletion utility.
ai (p93), AIPS++ code checkin utility.
alog (p100), AIPS++ change log reporting utility.
amv (p105), AIPS++ code rename utility.
au (p109), AIPS++ code update utility.
ax (p115), AIPS++ code deletion utility.
mktree (p126), create AIPS++ directory hierarchy.
rao (p133), invoke *ao* remotely via *ssh*(1).

Author

Original: 1992/04/28 by Mark Calabretta, ATNF.

9.5 *alog*

AIPS++ change log reporting utility.

Synopsis

```
alog [--archive]
   [--file=file | -f file]
   [--dorescind]
   [--listall]
   [--master | -m]
   [--package=package | -p package]
   [--noheader]
   [options for rlog]
   [file1 [file2 ...]]
```

Description

alog prints change log messages for AIPS++ source files. It is a front-end to the RCS *rlog* command.

If any files are specified then *alog* simply runs *rlog* on them filtering the output to remove RCS-specific information. The filenames may be specified via a relative or absolute pathname, with or without the RCS `,v` suffix.

If no files are specified on the command line, *alog* executes *rlog* for all AIPS++ sources, or for all sources within a particular AIPS++ (see §1.1.1, p1) package as specified by the *--package* option. In this instance, apart from filtering the output of *rlog* to remove RCS-specific information, *alog* also removes all references to files for which no revisions were selected. It thus produces a concise, formatted change log.

alog normally consults the RCS repository pointed to by the `$AIPSROOT/rcs` symbolic link, but can optionally be made to consult the master or archive if it is available (see §1.1.4, p5).

Options

The long form `POSIX` style options are provided to help differentiate options intended for *alog* from those it passes to *rlog*.

--archive Consult the archived RCS files. These include all RCS revisions up to the base release of the current major version. This option can only be used in Socorro.

--file=file | *-f* file The specified file, if it exists, contains a timestamp in *adate* (p51) format which *alog* converts to a *-d* option for *rlog*. The timestamp is then updated. This is useful for generating incremental change logs.

--dorescind Include rescinded files in the report (these only exist in the archive and master RCS repositories).

--listall List all files, the default behaviour is for ***alog*** to filter out files without revisions unless the *-R*, *-h*, or *-t* ***rlog*** options are specified.

--master* | *-m Consult the master RCS files. The AIPS++ master and slave RCS files contain only the current and preceeding major versions. The slave is typically a day to a week behind the master. This option should only be used in Socorro.

***--package*=package | *-p* package** Produce a report for the specified AIPS++ package only.

--noheader Don't add a separate header for each package.

Other options are passed directly to ***rlog*** without verification, check the manual page for ***rlog*** for an explanation. In particular, the *-d* option may be used to specify a range of dates, and *-w* to specify a list of users.

Notes

- If file name arguments are specified, ***alog*** will use the RCS symbolic links in the programmer directory created by ***mktree*** (p126).

Diagnostics

Status return values
  0: success
  1: initialization error

Examples

The following ***cron*** job is executed weekly on the AIPS++ master and the result e-mailed to interested parties:

```
# Report files currently checked out.
00 21 * * 0 (. $HOME/.profile ; alog -m -L -t) 2>&1 | \
   mail -s "AIPS++ sources currently checked out" aips2-workers
```

The *-L* and *-t* options are recognized by ***rlog***; *-L* causes it to ignore RCS files which have no locks set, and *-t* causes it to print the header information and descriptive text only (that is, basically everything except the change log entries). Much of the header information is then discarded by ***alog***.

See also

The unix manual page for ***cron***(1).
The manual page for ***rlog***(1), the RCS log command.
AIPS++ variable names (§1.2, p8).
AIPS++ code management configuration (§1.1.4, p5).
adate (p51), AIPS++ time reporting utility.
mktree (p126), create AIPS++ directory hierarchy.

Author

Original: 1993/01/22 by Mark Calabretta, ATNF.

9.6 *am*

Utility to change the descriptive text in an AIPS++ RCS version file.

Synopsis

am [-*au*] file1 [file2 ...]

Description

am changes the descriptive text in an existing AIPS++ RCS version file or files. It is a front-end for the RCS *rcs* command with option -*t*. The descriptive text will be obtained from `stdin`.

am assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). The filename arguments specified to *am* must be simple filenames without path or RCS (`,v`) suffix. *am* determines the corresponding master directory from the current working directory.

am modifies the master RCS version file by executing `$MSTRETCD/am_master`, where `$AIPSMSTR` is expected to be NFS mounted from the AIPS++ master host (see §1.1.4, p5).

Options

-*au* After modifying the master RCS version file, *am* copies it back to the local slave RCS repository (if it exists), giving it a file ownership defined by the owner of *am* (*aips2mgr* at the local site). *am* itself must be setuid to this account.

Notes

- The RCS *rcs* command must have write permission on the version file in order to modify it. The file ownership is defined by the owner of `$MSTRETCD/am_master` (*aips2adm*). This script must have the `setuid` bit set (see *chmod*).
- While *mktree* (p126) creates RCS symbolic links in the programmer directory hierarchy, these are not used by *am*. Their main function is to allow *gmake* (p154) to automatically check out the `makefile` for a directory. They also allow RCS utilities such as *alog* (p100) and *rlog* to be applied readily to the slave RCS repository. *am* does not make use of these RCS symbolic links but deals directly with the AIPS++ master repository by specifying absolute pathnames.
- Modifications are disabled if `$MSTRETCD/ai_disable` exists, see *ai* (p93). If this file contains any text it will be printed as an informative message.

Diagnostics

Status return values
`0`: success

1: initialization error
2: failed to update the master
3: updated master, but slave not updated

Examples

The command

```
cd $HOME/aips++/code/install/codemgmt
am -au am
```

would allow the descriptive text for *am* to be changed interactively.

Bugs

- *am* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The unix manual page for *chmod*(1).
AIPS++ variable names (§1.2, p8).
ae (p92), AIPS++ RCS semaphore file deletion utility.
ai (p93), AIPS++ code checkin utility.
alog (p100), AIPS++ change log reporting utility.
amv (p105), AIPS++ code rename utility.
au (p109), AIPS++ code update utility.
ax (p115), AIPS++ code deletion utility.
mktree (p126), create AIPS++ directory hierarchy.

Author

Original: 1995/03/17 by Mark Calabretta, ATNF.

9.7 *amv*

AIPS++ file and directory renaming utility.

Synopsis

amv [*-src* | *-top*] old new

Description

amv is used to rename AIPS++ files and directories. It does not do any renaming itself but instead appends renaming commands to the action list contained within ***ax_master*** (p115). It then invokes ***ax_master*** on the master AIPS++ RCS repository which must be available (see §1.1.4, p5). The renames will be applied to the slave RCS repositories and system by ***inhale*** (p78) and ***sneeze*** (p85) respectively. This mechanism allows AIPS++ slave sites to follow changes made to the master.

General users can rename directories at or below the third level below `$AIPSCODE` (see §1.1.1, p1). In particular, this includes module and application subdirectories. Renaming of higher-level directories is only allowed for members of the *aips2mgr* group.

Options

-src Rename files or directories in the AIPS++ RCS repository and code subtree. Both arguments must be specified as a pathname relative to `$AIPSCODE` (and therefore `$AIPSLAVE` and `$AIPSMSTR`. For example,

```
amv -src aips/implement/Foo.cc vlbi/implement/Bar.cc
```

would result in the following renames:

```
$AIPSMSTR/aips/implement/Foo.cc,v -> $AIPSMSTR/vlbi/implement/Bar.cc,v
$AIPSLAVE/aips/implement/Foo.cc,v -> $AIPSLAVE/vlbi/implement/Bar.cc,v
$AIPSCODE/aips/implement/Foo.cc   -> $AIPSCODE/vlbi/implement/Bar.cc
```

the first when ***ax_master*** (p115) is invoked by ***amv*** itself, and the other two when invoked by ***inhale*** (p78). The following deletions will also result when ***ax_master*** is invoked by ***sneeze*** (p85):

```
$AIPSARCH/lib/libaips.a(Foo.o)
$AIPSARCH/libdbg/libaips.a(Foo.o)
```

-top Rename a file or directory beneath `$AIPSROOT`. Both arguments must be specified as a pathname relative to `$AIPSROOT`. For example,

```
amv -top docs/README docs/ReadMe
```

would result in renaming

```
$AIPSROOT/docs/README  -> $AIPSROOT/docs/ReadMe
```

when ***sneeze*** (p85) invokes ***ax_master*** (p115). Note the following:

```
amv -top master/old master/new
amv -top slave/old slave/new
amv -top rcs/old rcs/new
```

The first only renames the master RCS version file, the second only renames the slave RCS version file, and by convention the third renames both.

Resources

- `account.manager`: Account name and group of the owner of the AIPS++ source code. ***amv*** only allows members of the *aips2mgr* group to delete higher-level directories. `$HOME/.aipsrc` is ignored.

Notes

- ***amv*** uses ***av*** (p111) to verify new filenames and creates addenda to the ***av*** filename cache for them.
- It is an error for the new name to refer to a file or directory which already exists - use ***ax*** (p115) to delete any such pre-existing file or directory if necessary.
- It is not permitted to rename a file or directory in the AIPS++ system areas. This must be done by using the ***-sys*** option to ***ax*** (p115) to delete the file or directory and relying on a system rebuild to recreate any missing files.
- When ***ax_master*** (p115) is invoked on the master RCS repositories it uses a temporary action list placed in `MSTRETCD/ax_list` by ***amv***. This mechanism prevents the same renames being applied by ***ax_master*** twice in the same version of AIPS++.
- Renames may be disabled by creating a file `$MSTRETCD/amv_disable`. If this file contains any text it will be printed as an informative message.
- ***amv*** must be setuid to *aips2mgr*.

Diagnostics

Status return values
- `0`: success
- `1`: initialization error
- `2`: usage error
- `3`: failed to get ***ax_master***

Examples

To move a module subdirectory from the `trial` package (see §1.1.1, p1) to the `aips` package use

```
amv -src trial/implement/Module aips/implement/Module
```

It may be convenient in *csh* and some other shells which allow it to use a command of the form

```
amv -src aips/implement\{,/FooMan}/Chu.cc
```

The shell would expand this to

```
amv -src aips/implement/Chu.cc aips/implement/FooMan/Chu.cc
```

Refer also to the examples provided in the "Options" section above.

Bugs

- *amv* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

AIPS++ code management configuration (§1.1.4, p5).
AIPS++ variable names (§1.2, p8).
inhale (p78), AIPS++ code import utility.
sneeze (p85), AIPS++ system rebuild utility.
av (p111), AIPS++ filename validation utility.
ax (p115), AIPS++ code deletion utility.

Author

Original: 1993/07/23 by Mark Calabretta, ATNF.

9.8 *asme*

Execute a command with the effective uid set to the real uid.

Synopsis

asme [command]

Description

asme executes a command with the effective uid set to the real uid. The requirement for such a utility arises because Bourne shell offers no mechanism within a `setuid` script to allow a command to be executed with the real uid.

Options

None.

Notes

- The command may be omitted.

Diagnostics

Status return values
`-1`: fork failed
`0`: success, value returned on `stdout`
`else`: exit status returned by the command

Examples

ai (p93) runs `setuid` to *aips2mgr*. It uses the following construct to ensure that the file to be checked in is readable by user and group:

```
asme "chmod ug+r $i"
```

See also

The unix manual page for *chmod*(1).
ai (p93) AIPS++ code checkin utility.

Author

Original: 1996/03/28 by Mark Calabretta, ATNF.

9.9 *au*

AIPS++ source code update utility.

Synopsis

```
au file1 [file2 ...]
```

Description

au updates the AIPS++ `slave` and `code` areas with the lastest version of a file in `$AIPSMSTR`.

au assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). The filename arguments specified to *au* must be simple filenames without path or RCS `,v` suffix. *au* determines the corresponding master directory from the current working directory.

au expects the master RCS repository, `$AIPSMSTR`, to be NFS mounted from the AIPS++ master host (see §1.1.4, p5). It copies the RCS version file from the master to the slave RCS repository, giving it a file ownership defined by the owner of *au* (aips2mgr at the local site). *au* must itself have the `setuid` bit set (see *chmod*).

If successful, *au* then checks out a plain-text copy of the code from the `$AIPSRCS` subdirectory into the corresponding `$AIPSCODE` subdirectory.

Newly created directories on the master may also be propagated to the local installation.

Programmers who have an account on the AIPS++ master host, *aips2.nrao.edu*, may instead use a related utility, *rau* (p136), which uses *ssh* to transfer the RCS version files in compressed form.

Programmers on a Linux or Digital UNIX platform, or on any other platform that does not support setuid shell scripts, should instead use a related utility, *sau* (p141). *sau* runs a non-setuid copy of *au*, *sau.sh*, via a C-based setuid wrapper program.

Options

None.

Diagnostics

Status return values
- 0: success
- 1: initialization error
- 2: master file not found
- 3: slave not updated
- 4: code not updated

Examples

The command

```
cd $HOME/aips++/code/install
au au
```

would update the slave and plain-text copies of ***au***, but not the script in the AIPS++ `bin` area.

Bugs

- ***au*** copies RCS version files directly from the master RCS repository, which may be NFS mounted from a machine on the other side of a slow internet link. The copy is handled more efficiently by ***rau*** (p136) which compresses the files for transfer.
- ***au*** requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The unix manual page for ***chmod***(1).
AIPS++ variable names (§1.2, p8).
ai (p93), AIPS++ code checkin utility.
amv (p105), AIPS++ code rename utility.
ao (p97), AIPS++ code checkout utility.
ax (p115), AIPS++ code deletion utility.
rau (p136), update AIPS++ sources from the master via ***ssh***(1).
sau (p141), AIPS++ code update utility.
mktree (p126), create AIPS++ directory hierarchy.

Author

Original: 1992/04/30 by Mark Calabretta, ATNF.

9.10 *av*

AIPS++ filename validation utility.

Synopsis

> *av* `filename1 [filename2,...]`

Description

av validates an AIPS++ class implementation filename against those already in the system to ensure uniqueness in the first 15 characters. It consults a cache of AIPS++ pathnames maintained by other AIPS++ code management utilities (***ai*** (p93), ***amv*** (p105), ***exhale*** (p73)). Any matches found are printed on stdout.

Options

None.

Notes

- The pathname cache is stored in compressed form using ***gzip*** to minimize the amount of data which must be fetched from the master during a checkin or rename.
- If several filenames are to be validated it is more efficient to run ***av*** once with multiple filename arguments than to run it separately for each filename.

Diagnostics

Status return values
`0`: success
`1`: initialization error
`2`: cache file not accessible

Examples

The command

```
av BaseMappedArrayIter
```

would print the string

```
aips/implement/Tables/BaseMappedArrayEngine
```

indicating that `BaseMappedArrayIter` would not be acceptable as a new file name because the first 15 characters conflict with `BaseMappedArrayEngine`.

Files

`$MSTRETCD/av_cache.gz` ...pathname cache.

See also

The GNU manual page for ***gzip*** (1).
AIPS++ variable names (§1.2, p8).
ai (p93), AIPS++ code checkin utility.
amv (p105), AIPS++ code rename utility.
exhale (p73), AIPS++ code export utility.

Author

Original: 1994/11/11 by Mark Calabretta, ATNF.

9.11 *avers*

AIPS++ version reporting utility.

Synopsis

avers [-*b* | -*l*]

Description

avers reports version information for an AIPS++ installation.

The AIPS++ version consists of three fixed-length fields of the form `MM.NNN.PP`. The first field of two digits is the major version number, the second field of three digits is the minor version number, and the third field of two digits is the patch level.

The major version number is incremented whenever a new base release is produced. A base release is the publically distributed production code. The minor version number is incremented on each cycle of the code distribution system for AIPS++ development sites (see ***exhale*** (p73) and ***inhale*** (p78)), and the patch version distinguishes different patch levels applied to the base release for the public distribution.

The long report contains the following version and timing information:

`Master:` Version last fetched by ***inhale*** (p78).
`Slave:` Version and time of completion of last slave update.
`Code:` Version and time of completion of last code update.
`System:` Version and time of completion of last system rebuild.
`Latest:` Version currently available from the master.
`Time:` The current time, in AIPS++ format (see ***adate*** (p51)).

Each of the installation steps is tagged as being *base*, *incremental*, or *cumulative*. The short report only includes the version last fetched by ***inhale*** (p78), and without the "`Master:`" prefix.

Options

-*b* Report the version last fetched by ***inhale*** (default).

-*l* Report the version last fetched by ***inhale***, the time at the completion of each stage of its installation, the latest version available from the master, and the current time.

Diagnostics

Status return values
`0`: success
`1`: invalid option
`2`: `$AIPSPATH` not defined

Examples

Sample output from the command

```
avers -l
```

appears as follows

```
Master: 01.021.00 Wed 1992/09/16 10:30:32 GMT
 Slave: 01.021.00 Wed 1992/09/16 12:01:04 GMT  (incremental)
  Code: 01.021.00 Wed 1992/09/16 12:03:08 GMT  (incremental)
System: 01.021.00 Wed 1992/09/16 12:07:55 GMT  (incremental)
Latest: 01.022.00 Wed 1992/09/16 22:30:38 GMT
  Time:           Thu 1992/09/17 04:10:42 GMT
```

See also

adate (p51), AIPS++ time reporting utility.
exhale (p73), AIPS++ code export utility.
inhale (p78), AIPS++ code import utility.

Author

Original: 1992/09/17 by Mark Calabretta, ATNF.

9.12 *ax*

AIPS++ deletion utility.

Synopsis

ax [*-src* | *-sys* | *-top*] target1 [target2 ...] [...]

Description

ax is used to delete AIPS++ files and directories. It does not delete them itself but instead appends deletion commands to the action list contained within ***ax_master*** (see below). It then invokes ***ax_master*** on the master AIPS++ RCS repository which must be available (see §1.1.4, p5). The deletions will be applied to the slave RCS repositories and system by ***inhale*** (p78) and ***sneeze*** (p85) respectively. This mechanism allows AIPS++ slave sites to follow changes made to the master.

General users can delete directories at or below the third level below `$AIPSCODE` (see §1.1.1, p1). In particular, this includes module and application subdirectories. Deletion of higher-level directories is only allowed for members of the *aips2mgr* group.

ax_master does not actually delete anything from the AIPS++ master RCS repositories but instead "rescinds" them by renaming them so as to hide them from the code distribution system, see ***exhale*** (p73). Rescinded files and directories are given names of the form `.MM.NNN.filename`, where `MM.NNN` is the AIPS++ version number at the time the file is rescinded (see ***avers*** (p113)).

Options

-src Delete a pair of files from the AIPS++ RCS repository and code subtree. The target must be specified as a pathname relative to `$AIPSCODE` (and therefore `$AIPSLAVE` and `$AIPSMSTR`, and need not have an RCS `,v` suffix. For example,

```
ax -src aips/apps/App1/App1.cc
```

would cause the following files to be deleted

```
$AIPSMSTR/aips/apps/App1/App1.cc,v
$AIPSLAVE/aips/apps/App1/App1.cc,v
$AIPSCODE/aips/apps/App1/App1.cc
```

the first is rescinded when ***ax_master*** is invoked by ***ax*** itself, and the other two are deleted when ***ax_master*** is invoked by ***inhale*** (p78). Note, however, that any executables associated with an application will not be deleted. These may be deleted explicitly by using the *-sys* option (see below).

If the target specified is a class implementation file, the object module will be deleted from the relevant library when ***ax_master*** is invoked by ***sneeze*** (p85). For example,

```
ax -src aips/implement/Class.cc
```

would result in the following deletions

```
$AIPSMSTR/aips/implement/Class.cc,v
$AIPSLAVE/aips/implement/Class.cc,v
$AIPSCODE/aips/implement/Class.cc
$AIPSARCH/lib/libaips.a(Class.o)
$AIPSARCH/libdbg/libaips.a(Class.o)
```

If necessary, *ax_master* will also *ranlib* the object libraries and rebuild any sharable objects.

Directories specified as targets will be rescinded from the master and recursively deleted from the slave and code areas.

-sys Delete a file from the AIPS++ system area. The target must be specified as a pathname relative to `$AIPSARCH` - the master, slave, and code areas are not affected. For example,

```
ax -sys bin/Application bindbg/Application
```

would mark

```
$AIPSARCH/bin/Application
```

and

```
$AIPSARCH/bindbg/Application
```

for deletion by *ax_master* when invoked by *sneeze* (p85).

Targets specified in the form `x(y)` denote library modules. For example,

```
ax -sys lib/libaips.a(Class.o)
```

would mark `Class.o` for deletion from

```
$AIPSARCH/lib/libaips.a
```

and

```
$AIPSARCH/libdbg/libaips.a
```

by *ax_master* when invoked by *sneeze* (p85). If necessary, *ax_master* would then *ranlib* the object libraries and rebuild any sharable objects.

A directory may be specified as the target, in which case it will be recursively deleted.

-top Delete a file beneath `$AIPSROOT`. The target must be specified as a pathname relative to `$AIPSROOT`. This is a catch-all target and should only be used when the ***-src*** and ***-sys*** options are inappropriate.

If the target refers to a file in the master repository then it will be rescinded when ***ax_master*** is invoked by ***ax***. If it refers to a file in the slave or code areas then it will be deleted when ***ax_master*** is invoked by ***inhale*** (p78). If it refers to a file in the system area then it will be deleted when ***ax_master*** is invoked by ***sneeze*** (p85) running on a machine of the corresponding architecture.

For example,

```
ax -top docs/reference/Example.ps
```

would mark

```
$AIPSROOT/docs/reference/Example.ps
```

for deletion by ***ax_master*** when invoked by ***inhale*** (p78). Note the following:

```
ax -top master/aips/apps/App1/App1.cc,v
ax -top slave/aips/apps/App1/App1.cc,v
ax -top rcs/aips/apps/App1/App1.cc,v
```

The first rescinds the master RCS version file, the second deletes the slave RCS version file, and by convention the third does both. Note that the RCS `,v` suffix does need to be specified here.

A directory may be specified as the target, in which case it will be rescinded from the master and recursively deleted from the slave and code areas.

Resources

- `account.manager`: Account name and group of the owner of the AIPS++ source code. ***ax*** only allows members of the *aips2mgr* group to delete higher-level directories. `$HOME/.aipsrc` is ignored.

Notes

- Manual invokation of ***ax_master*** is forbidden. However, it may be of interest to know that it has a mode which causes it to apply all entries in its action list for the current major AIPS++ version. This is used by ***inhale*** (p78) and ***sneeze*** (p85) immediately before downloading a cumulative update.
- When ***ax_master*** is invoked on the master RCS repositories it uses a temporary action list placed in `$MSTRETCD/ax_list` by ***ax***. This mechanism prevents the same file from being rescinded twice from the master by ***ax_master*** in the same version of AIPS++.
- ***ax*** may be disabled by creating a file `$MSTRETCD/ax_disable`. If this file contains any text it will be printed as an informative message to anyone trying to use ***ax***.

  ax must be owned by *aips2mgr* and have the `setuid` bit set.

Diagnostics

Status return values
- 0: success
- 1: initialization error
- 2: failed to get *ax_master*
- 3: argument error

Examples

To recursively delete module subdirectory `trial/implement/Junk` use

```
ax -src trial/implement/Junk
```

Where several files are to be deleted, it may be convenient in *csh* and some other shells which allow it to use a command of the form

```
cd /aips++/code
ax -src aips/implement/\{Foo,Man,Chu}.*
```

Assuming the files existed, the shell would expand the argument to

```
aips/implement/Foo.cc aips/implement/Foo.h \
aips/implement/Man.cc aips/implement/Man.h \
aips/implement/Chu.cc aips/implement/Chu.h \
```

Note that the '`cd /aips++/code`' provided the context for the shell's wildcarding mechanism.

Refer also to the examples provided in the "Options" section above.

Bugs

- *ax* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

AIPS++ variable names (§1.2, p8).
getrc (p43), query AIPS++ resource database.
exhale (p73), AIPS++ code export utility.
inhale (p78), AIPS++ code import utility.
ai (p93), AIPS++ code checkin utility.
amv (p105), AIPS++ code rename utility.
ao (p97), AIPS++ code checkout utility.
au (p109), AIPS++ code update utility.
avers (p113), AIPS++ version report utility.

Author

Original: 1992/05/14 by Mark Calabretta, ATNF.

9.13 *buildchangelog*

Process AIPS++ changelog files and transform them to html files containing the selected changes.

Synopsis

> *buildchangelog* [*-dir[ectory]* directory] [*-split[file]* area|package|module]
> [select-options <directory>

Description

All AIPS++ changelog files found in the given directory and its subdirectories are processed and the selected changes are written to html files. Also a few html files containing an overview of the changes are generated with links to the detailed html files.
The given directory has to be a code directory; i.e. it has to end in `/code`.
The changes can be selected in various ways using the command options. By default the changes in the currently developed release are selected.
The output directory can be specified in the options, which defaults to the working directory.

buildchangelog is run by the *sneeze* (p85) script if the appropriate *makedefs* (p155) variable is set. In that case it builds html files for the changes in the currently developed release. The AIPS++ documentation pages have links to the overview files generated in this way, so at all times the achievements in the currently developed release can be seen on the AIPS++ web pages.

One can also use *buildchangelog* to find, for example, all changes made to glish in the last year.

Options

-dir[ectory] **directory** The directory of the generated html files. Default is the working directory.

-split[file] **value** This case insensitive option determines how many html files are written. If not given, all changes are combined in one html file.
-split area generates an html file per area with name `changelog_<area>.html`.
- split package generates an html file per package with name
`changelog_<area>_<package>.html`.
-split module generates an html file per module with name
`changelog_<area>_<package>_<module>.html`.
If the split option is given (and not blank), a summary file per area ia generated with name `changelog_<area>_summary.html`.
In any case an overview file is generated with name `changelog_summary.html`.

-date **ranges** Select changes matching the given date range(s). Multiple ranges can be given separated by commas. Each range is a start date optionally followed by a colon and an end date. Each date can be given as yyyy-mm-dd or as dd-mm-yyyy, but an end date has to be given in the same order as its start date. The month can be given as a number or

as a 3 letter case insensitive name. It is not necessary to give all parts of a date, so it is possible to select a given year or range of years (or days or months for that matter). Again, an end date and start date have to match in that respect.
For example,

```
-date 2000
-date 1-Jan-2000:31-Dec-2000
-date 1995,1996,1997,1998,1999,2000
-date 1995:2000
```

The first 2 examples are equivalent; select all of the year 2000.
The latter 2 examples are also equivalent; select all of years 1995 to 2000 (inclusive).

-nodate **ranges** Select all changes NOT matching the given date range(s).

-id **ranges** Select all changes matching the given id range(s). Multiple ranges can be given separated by commas. Each range is a start id optionally followed by a colon and an end id.
For example,

```
-id 1:10,20,24:30
```

-noid **ranges** Select all changes NOT matching the given id range(s).

-area **names** Select all changes matching the given area name(s). Multiple areas can be given separated by commas. Note that the names are case sensitive.
For example,

```
-area System,Glish,Tool,Library,LearnMore
```

This eaxmple shows all available areas.

-noarea **names** Select all changes NOT matching the given area name(s).

-package **names** Select all changes matching the given package name(s). Multiple packages can be given separated by commas. Note that the names are case sensitive.
For example,

```
-package aips
```

Note that only areas Library and Tool have multiple packages.

-nopackage **packages** Select all changes NOT matching the given package(s).

-module **names** Select all changes matching the given module (or tool) name(s). Multiple modules can be given separated by commas. Note that the names are case sensitive.
For example,

```
-module Tables,Arrays
```

Note that the word module is also used for the tool name, so a specific tool is selected in the same way.

-nomodule **names** Select all changes NOT matching the given module name(s).

-avers **ranges** Select all changes matching the given AIPS++ version range(s). Multiple ranges can be given separated by commas. Each range is a start version optionally followed by a colon and an end version. Note that the version is treated as a float number (like 1.4).
By default only the currently developed AIPS++ release is selected. Therefore the special (case insensitive) value ***all*** can be given to indicate that all versions have to be selected.
For example,

```
-avers all
-avers 1.4:1.9
```

-noavers **ranges** Select all changes NOT matching the given AIPS++ version range(s).

-type **types** Select all changes matching the given type(s). Multiple types can be given separated by commas. Note that the type names are case insensitive.
For example,

```
-type code,documentation,test,review
```

This example shows all available types.

-notype **types** Select all changes NOT matching the given type(s).

-category **categories** Select all changes matching the given categories. Multiple categories can be given separated by commas. Note that the category names are case insensitive.
For example,

```
-category new,change,bugfix,removed,other
```

This example shows all available categories.

-nocategory **categories** Select all changes NOT matching the given categories.

Notes

- The change entries have an area, package, and module component. There are a few areas:

  Library for all changes in the library code.

  Tool for all changes in the AIPS++ tools (applications).

  Glish for all changes in glish.

  System for all changes in the system area (directory install).

  LearnMore for all changes in notes, papers, etc. (directory doc).

Diagnostics

Status return values
  0: success
  1: invalid option given

Examples

The command

```
buildchangelog -split module
  -dir $(AIPSROOT)/docs/project/releasenotes/changelogs
  $(AIPSROOT)/code
```

is used in the code makefile to generate the html files when building the system.

```
buildchangelog -avers all -area Glish /aips++/code
```

can be used to find all changes ever made to glish. It will generate the files `./changelog.html` and `./changelog_summary.html`.

Bugs

- *buildchangelog* does not check if a change entry is filled in correctly.

See also

ac (p89) AIPS++ changelog maintenance.

Author

Original: 2000/06/05 by Ger van Diepen, NFRA

9.14 *doover*

Generate a function which returns AIPS++ version information.

doover

Description

doover generates C++ code which records version information found in `$AIPSCODE/VERSION`. It is intended to be used only by the AIPS++ `makefiles` (p162).

The C++ code defines the following external variables

- `const int aips_major_version;` The AIPS++ major version number (see *avers* (p113)).
- `const int aips_minor_version;` The AIPS++ minor version number.
- `const int aips_patch_version;` The AIPS++ patch version number.
- `const char* aips_version_date;` The date on which this version was produced in AIPS++ date format (see *adate* (p51)).
- `const char* aips_version_info;` Any identifying version information.

and it also defines the following global function which reports the version information in a standard format:

```
void report_aips_version(ostream &os)
```

Executables can access these external variables by virtue of declarations made in `aips.h`.

Options

None.

Notes

- `makefile.imp` (p162) uses *doover* to check and if necessary generate `$LIBDBGD/version.o` and `$LIBOPTD/version.o` whenever it builds a system object library. The `version.o` object modules are dependent on the `$(AIPSCODE)/VERSION` file. They are not inserted into the object library since that would cause all executables to be rebuilt.

Diagnostics

Status return values
`0`: success
`1`: initialization error
`2`: `VERSION` file not found

Examples

Typing

```
void report_aips_version(ostream &os)
```

shows the code containing the version information derived from the current `$(AIPSCODE)/VERSION`.

See also

AIPS++ variable names (§1.2, p8).
adate (p51), AIPS++ time reporting utility.
avers (p113), AIPS++ version report utility.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

Author

Original: 1996/02/29 by Mark Calabretta, ATNF.

9.15 *mktree*

Create an AIPS++ directory hierarchy with RCS symlinks.

Synopsis

mktree [-*d*] [-*l*] [-*r*] [-*s*]

Description

mktree is an interactive utility used by AIPS++ programmers to create a local workspace. It works incrementally in that it won't recreate directories or symbolic links which already exist, but may create new ones which have appeared since it was first run.

mktree creates a directory tree which shadows the `$AIPSCODE` subdirectory tree (see §1.1, p1). In each directory it creates an `RCS` symbolic link to the corresponding RCS directory under `$AIPSRCS`, thereby creating a window into the local AIPS++ code repository.

The main function of the `RCS` symlinks is to provide an access route for *gmake* (p154) to the AIPS++ `makefiles` (p162). Although code checkin and checkout must be done using *ai* (p93) and *ao* (p97) which do not use the `RCS` symlinks, the symlinks do allow other RCS utilities such as *rlog* and *rscdiff* to be used as though the AIPS++ RCS repositories existed in the user's own workspace. *ao* also allows `ao RCS/*,v` to check out everything in a directory.

mktree also creates the `$CODEINCD` directory and symbolic links therein. These allow include files to be specified in the form `#include <aips/Class.h>` for example (see §1.1.1, p1).

mktree can be used to create or update a limited portion of the directory hierarchy. If the current working directory is a subdirectory of `*/code`, then only that portion of the tree will be created (see the example below).

mktree checks for defunct directories and will delete them and the files therein after seeking confirmation (unless the -*s* option was specified).

Options

- -*d* Delete any pre-existing symbolic links. This can be used to force symlinks to be recreated in the uncommon situation where a pre-existing programmer tree must be updated for a changed `AIPSROOT`, and where the old `AIPSROOT` still remains. (Note that if the old `AIPSROOT` did not still remain then the symlinks would be recreated without needing to use this option.)
- -*l* `RCS` symbolic links are normally created only if `$AIPSRCS` is being shadowed (see the -*r* option below), or if the `$AIPSCODE/rcs` symlink exists. This option forces them to be created regardless.
- -*r* Shadow the `$AIPSRCS` tree rather than `$AIPSCODE`. This is used by *inhale* (p78) to construct or update the `$AIPSCODE` tree at consortium sites.
- -*s* Don't ask for confirmation of the parent directory, *or when deleting defunct directories and the files therein.* Use with caution!

A *-master* option which creates RCS symlinks directly into $AIPSMSTR is also recognized but is only intended for use by *exhale* (p73) when building a new public release.

Notes

- *mktree* ignores tmplinst directories, both in creating the tree and when deleting defunct directories.

Diagnostics

Status return values
0: success
1: initialization error

Examples

To construct an AIPS++ workspace from scratch or update an existing one

```
mkdir $HOME/aips++
cd $HOME/aips++
mktree
```

This is equivalent to

```
mkdir $HOME/aips++/code
cd $HOME/aips++/code
mktree
```

To just create or update the aips package (see §1.1.1, p1)

```
cd $HOME/aips++/code/aips
mktree
```

See also

The manual page for *rlog*(1).
The manual page for *rcsdiff*(1).
AIPS++ variable names (§1.2, p8).
ai (p93), AIPS++ code checkin utility.
alog (p100), AIPS++ change log reporting utility.
amv (p105), AIPS++ code rename utility.
ao (p97), AIPS++ code checkout utility.
au (p109), AIPS++ code update utility.
ax (p115), AIPS++ code deletion utility.

Author

Original: 1992/03/07 by Peter Teuben, BIMA.

9.16 *rac*

rac is effectively the same as *ac* (p89). The only difference is that it is using *rao* (p133) and *rai* (p130) instead of *ao* (p97) and *ai* (p93) to be able to update the changelog from a remote site.

Synopsis

rac [options for *rac*]

Description

You are referred to the description of *ac* (p89).

See also

rai (p130), AIPS++ code checkin utility.
rao (p133), AIPS++ code checkout utility.
buildchangelog (p120) process AIPS++ changelog files.

Author

Original: 2000/06/05 by Ger van Diepen, NFRA.

9.17 *rai*

Check in AIPS++ sources by invoking *ai* (p93) remotely via *ssh*.

Synopsis

rai [options for *ai*]

Description

rai is an *ssh*-based front-end to *ai* (p93). It checks in files by transmitting them to the AIPS++ master host via *ssh* as a *gzip*'d tar file and running *ai* there. This is provided as an alternative to invoking *ai* directly on a machine on which the AIPS++ master directory is remotely NFS mounted.

In order to use *rai* a programmer must have an account on *aips2.nrao.edu* and it must be set up for *ssh* access from the local host via a suitable entry in the `.ssh/authorized_keys` file (`.ssh/authorized_keys2` for ssh2 based access). The programmer must also have an up-to-date AIPS++ workspace on *aips2.nrao.edu* and it *must* be rooted at `$HOME/aips++` (see *mktree* (p126)). Visit the Using SSH page for more details on using ssh in AIPS++.

Options

Refer to the options for *ai* (p93). *rai* itself has no options.

Notes

- *rai* handles directories as well as files.
- *rai* is faster than *ai* because
  - It only has to do one file transfer rather than two.
  - It transfers files in compressed form.
  - Although it needs to invoke *ssh* two or three times the overhead involved in this is less than that of one automount of a remote filesystem.
- When accessing *aips2.nrao.edu* via *ssh* the programmer's login initialization scripts (e.g. `.bashrc`, `.cshrc`, `.esrc`, `.login`, `.profile`, `.rcrc`, `.tcshrc`) must not consume `stdin` or write to `stdout` or `stderr`.
- Where the account name on the *aips2.nrao.edu* differs from the local account name an environment variable may be used to translate between them:

```
AUID="<remotename>"
```

The `.ssh/authorized_keys` (`.ssh/authorized_keys2` for ssh2) file for the *remotename* account on *aips2.nrao.edu* should have an entry for each machine you may want to use remotely.

you will need to run ssh-keygen and put the generated key in the `.ssh/authorized_keys` on *aips2.nrao.edu*

- ***rai*** is not failsafe in the way that ***ai*** is failsafe. It has no way of determining whether a checkin succeeded and so provides no guarantees one way or the other. This arises because of the difficulty in recovering status returns from complicated command sequences executed by ***ssh***.

  If a checkin fails the error message from ***ci*** will be visible on `stderr` but ***rai*** will not catch an error return. The only course of action would be to log in to *aips2.nrao.edu* and clean up manually.

  For this reason, ***rai*** *never deletes files from the local workspace* but simply removes the write permission from them (unless invoked with the ***-l*** option) - *even if the checkin failed.* It is left to the programmer to decide whether the checkin succeeded and delete them explicitly.

  rai does delete files from the *aips2.nrao.edu* workspace if the checkin succeeded.

- ***rai*** exhibits the following differencees from ***ai*** by virtue of the fact that it runs ***ai*** remotely through an ***ssh***:

  - When ***ci*** says to

    ```
    enter description, terminated with single '.' or end of file:
    ```

    you must terminate with a single "." and not use `^D`. The reason is simply that `^D` means "end-of-file" to the ***ssh*** as well as ***ci***, so if you use `^D` when checking in multiple files ***ci*** will not accept change messages for succeeding files.
  - There may be noticable delays in entering the change message.

- ***ssh*** is used as the transfer agent rather than ***ftp*** since the latter requires a password to be supplied for each remote access.

Diagnostics

Status return values
`0`: success (returned from ***ai_master***)
`1`: initialization error, no checkin

Examples

The command

```
rai -au Foo.cc
```

would check in `Foo.cc` from the current directory into the corresponding AIPS++ master repository, updating the slave and plain-text copies in the process. The copy of `Foo.cc` in the current directory would *not* be deleted but it's write permission would be removed. If the checkin succeeded it would be appropriate to manually delete the file

```
rm Foo.cc
```

If the checkin failed then it would be necessary to log in to *aips2.nrao.edu* and recover manually.

Bugs

- Not all of the ***ai*** (p93) options are supported yet due to setuid issues.

See also

The manual page for ***ci***(1), the RCS checkin command.
The GNU manual page for ***gzip***(1).
The unix manual page for ***ftp***(1).
The unix manual page for ***ping***(1).
The unix manual page for ***ssh***(1).
ai (p93), AIPS++ code checkin utility.
ao (p97), AIPS++ code checkout utility.
au (p109), AIPS++ code update utility.
mktree (p126) create AIPS++ directory hierarchy.
rao (p133), invoke ***ao*** remotely via ***ssh***(1).
rau (p136), update AIPS++ sources from the master via ***ssh***(1).

Author

Original: 1996/08/06 by Mark Calabretta, ATNF.

9.18 *rao*

Check out AIPS++ sources by invoking *ao* (p97) remotely via *ssh*.

Synopsis

rao [options for *ao*]

Description

rao is an *ssh*-based front-end to *ao* (p97). It checks out files by running *ao* on the AIPS++ master host and transmitting them to the local host via *ssh* as a *gzip*'d tar file. This is provided as an alternative to invoking *ao* directly on a machine on which the AIPS++ master directory is remotely NFS mounted.

In order to use *rao* a programmer must have an account on *aips2.nrao.edu* and it must be set up for *ssh* access from the local host via a suitable entry in the `.ssh/authorized_keys` file (`.ssh/authorized_keys2` for ssh2). The programmer must also have an up-to-date AIPS++ workspace on *aips2.nrao.edu* and it *must* be rooted at `$HOME/aips++` (see *mktree* (p126)). Visit the Using SSH page for more details on using ssh in AIPS++.

Options

Refer to the options for *ao* (p97). *rao* itself has no options.

Notes

- *rao* handles directories as well as files.
- *rao* is faster than *ao* because
  - It only has to do one file transfer rather than two.
  - It transfers files in compressed form.
  - The overhead involved in the invokation of *ssh* is much less than that of one automount of a remote filesystem.
- When accessing *aips2.nrao.edu* via *ssh* the programmer's login initialization scripts (e.g. `.bashrc`, `.cshrc`, `.esrc`, `.login`, `.profile`, `.rcrc`, `.tcshrc`) must not consume `stdin` or write to `stdout` or `stderr`.
- Where the account name on the *aips2.nrao.edu* differs from the local account name an environment variable may be used to translate between them:

```
AUID="<remotename>"
```

The `.ssh/authorized_keys` file (`.ssh/authorized_keys2` for ssh2) for the *remotename* account on *aips2.nrao.edu* should have an entry for each machine you may want to use remotely.

you will need to run ssh-keygen and put the generated key in the `.ssh/authorized_keys` on *aips2.nrao.edu*

- *ssh* is used as the transfer agent rather than *ftp* since the latter requires a password to be supplied for each remote access.
- *rao* will prompt the user if writable copies of the files being checked out exist, allowing the user to choose between replacing them and aborting the check-out. Each file will be prompted for separately.
- *rao* will print a warning message when it removes locally-unwritable (read-only) copies of files in order to replace them with the checked-out copies.

Diagnostics

Status return values
 `0`: success (returned from *ao_master*)
 `1`: initialization error, no checkout

Examples

The command

```
rao -l Foo.cc
```

would check out `Foo.cc` with an exclusive lock. Then

```
rao -u Foo.cc
```

would remove the lock without checking the file in or deleting the working copy. The command

```
rao -l RCS/*,v
```

would check out everything in the directory (that is, assuming that `RCS` is a symlink into the slave directory, all files currently in the slave directory will be checked out of the master).

See also

The manual page for *co*(1), the RCS checkout command.
The GNU manual page for *gzip*(1).
The unix manual page for *ftp*(1).
The unix manual page for *ping*(1).
The unix manual page for *ssh*(1).

ai (p93), AIPS++ code checkin utility.
ao (p97), AIPS++ code checkout utility.
au (p109), AIPS++ code update utility.
mktree (p126) create AIPS++ directory hierarchy.
rai (p130), invoke ***ai*** remotely via ***ssh***(1).
rau (p136), update AIPS++ sources from the master via ***ssh***(1).

Author

Original: 1996/08/06 by Mark Calabretta, ATNF.

9.19 *rau*

Update AIPS++ sources from the master via *ssh*.

Synopsis

rau `file1 [file2 ...]`

Description

rau updates the AIPS++ `slave` and `code` areas with the lastest version of a file in `$AIPSMSTR`. It is provided as an alternative to *au* which operates on machines on which the AIPS++ master directory is remotely NFS mounted.

In order to use *rau* a programmer must have an account on *aips2.nrao.edu* and it must be set up for *ssh* access from the local host via a suitable entry in the `.ssh/authorized_keys` file (`.ssh/authorized_keys` for ssh2). The programmer must also have an up-to-date AIPS++ workspace on *aips2.nrao.edu* and it *must* be rooted at `$HOME/aips++`. *rau* assumes that you are working within a private directory tree which shadows that of the AIPS++ master source code tree, see *mktree* (p126). The filename arguments specified to *rau* must be simple filenames without path or RCS `,v` suffix. *rau* determines the corresponding master directory from the current working directory.

rau transmits files from the master host to the local host via *ssh* as a *gzip*'d tar file. It gives them a file ownesship defined by the owner of *rau* (aips2mgr at the local site). *rau* must itself have the `setuid` bit set (see *chmod*).

If successful, *au* then checks out a plain-text copy of the code from the `$AIPSRCS` subdirectory into the corresponding `$AIPSCODE` subdirectory.

Newly created directories on the master may also be propagated to the local installation.

Programmers on a Linux or Digital UNIX platform, or on any other platform that does not support setuid shell scripts, should instead use a related utility, *sau* (p141). *sau* runs a non-setuid copy of *au*, *sau.sh*, via a C-based setuid wrapper program.

Options

None.

Notes

- *rau* handles directories as well as files.
- *rau* is faster than *au* because
  - It transfers files in compressed form.
  - The overhead involved in the invokation of *ssh* is much less than that of one automount of a remote filesystem.

- When accessing *aips2.nrao.edu* via *ssh* the programmer's login initialization scripts (e.g. `.bashrc`, `.cshrc`, `.esrc`, `.login`, `.profile`, `.rcrc`, `.tcshrc`) must not consume `stdin` or write to `stdout` or `stderr`.
- Where the account name on the *aips2.nrao.edu* differs from the local account name an environment variable may be used to translate between them:

  ```
  AUID="<remotename>"
  ```

  The `.ssh/authorized_keys` file (`.ssh/authorized_keys2` for ssh2) for the *remotename* account on *aips2.nrao.edu* should have an entry for each machine you may want to use remotely.

  you will need to run ssh-keygen and put the generated key in the `.ssh/authorized_keys` on *aips2.nrao.edu*
- *ssh* is used as the transfer agent rather than *ftp* since the latter requires a password to be supplied for each remote access.

Diagnostics

Status return values
  `0`: initialization succeeded
  `1`: initialization error

Examples

The command

```
cd $HOME/aips++/code/install
rau rau
```

would update the slave and plain-text copies of *rau*, but not the script in the AIPS++ `bin` area.

Bugs

- *rau* requires that the host operating system support setuid shell scripts. Some operating systems, most notably Linux and Digital UNIX, do not support setuid shell scripts for security reasons.

See also

The manual page for *co*(1), the RCS checkout command.
The GNU manual page for *gzip*(1).
The unix manual page for *ftp*(1).
The unix manual page for *ping*(1).
The unix manual page for *ssh*(1).

ai (p93), AIPS++ code checkin utility.
ao (p97), AIPS++ code checkout utility.
au (p109), AIPS++ code update utility.
mktree (p126) create AIPS++ directory hierarchy.
rai (p130), invoke *ai* remotely via *ssh*(1).
rao (p133), invoke *ao* remotely via *ssh*(1).
sau (p141), AIPS++ code update utility.

Author

Original: 1996/08/16 by Mark Calabretta, ATNF.

9.20 *rcscat*

Concatenate two RCS version files.

Synopsis

rcscat `rcsfile rcsarkv`

Description

rcscat is a special-purpose utility which concatenates the archived RCS version file to the current RCS version file for an AIPS++ source. It is used by *exhale* (p73) to update the AIPS++ RCS archive when creating a new base release.

The first file argument specifies the master RCS file which *exhale* (p73) has just "upreved" to the new base release, *m*.0. It contains all revisions from *l*.0 to *m*.0, where $l = m$ - 1.

The second file argument specifies the archive RCS version file which contains all revisions from `1.0` to *l*.0.

The concatenated RCS version file which *rcscat* writes to `stdout` contains all revisions from `1.0` to *m*.0. *exhale* (p73) makes this the new archive RCS file.

Options

None.

Notes

- *rcscat* is not intended for general use.
- The process of producing a new base release is described in detail in the entry for *exhale* (p73).
- *rcscat* is implemented as a C program. It manipulates the internal contents of the RCS version files, relying on the fact that revision *l*.0 is identical to the preceding revision.
- *rcscat* can handle binary as well as plain-text sources.

Diagnostics

Status return values
`0`: success
`1`: usage error
`2`: file access error
`3`: RCS file format error

See also

The manual page for *rcsfile*(5), RCS version file format.
exhale (p73), AIPS++ code export utility.

Author

Original: 1995/07/20 by Mark Calabretta, ATNF

9.21 *sau*

AIPS++ source code update utility.

Synopsis

sau `file1 [file2 ...]`

Description

sau updates the AIPS++ `slave` and `code` areas with the lastest version of a file in `$AIPSMSTR`.

sau is a replacement for ***au*** (p109) (and ***rau*** (p136)) on operating systems, most notably Linux and Digital UNIX, that do not support setuid shell scripts. *sau* is a C-based setuid wrapper that runs a non-setuid copy of ***au*** (p109), *sau.sh*.

Refer to the ***au*** (p109) documentation for further information.

Options

None.

Diagnostics

Status return values
`0`: success
`-1`: exec of *sau.sh* failed.
Refer to the ***au*** (p109) documentation for other return values.

Examples

The command

```
cd $HOME/aips++/code/install
sau au
```

would update the slave and plain-text copies of ***au***, but not the script in the AIPS++ `bin` area.

See also

au (p109), AIPS++ code update utility.

Author

Original: 1997/09/08 by Jeff Uphoff, NRAO

9.22 *squiz*

Search for and examine AIPS++ sources.

Synopsis

squiz [-*c* category] [-*e*|-*E* expression] [-*l*] [-*F*] [-*p* package] [file]

Description

squiz searches for AIPS++ source files and optionally searches for strings inside the files it finds. Wildcarding is allowed in the file specification and the search is case insensitive by default.

If no file is specified the default is to search for (i.e. list) all files.

Options

-*c* **category** Restrict the search to a certain category:

- `application`, `apps`, `app`: Application source files.
- `fortran`, `for`, `ftn`: FORTRAN sources.
- `implement`, `imp`: Class implementation files.
- `test`, `tst`: Test programs.

The default is to search everything.

-*e* **expression** Perform a case insensitive search for the specified expression in all selected files. *grep*-type regular expressions are allowed. See also the -*l* option.

-*E* **expression** Case sensitive version of the -*e* option.

-*F* Do a case sensitive search for the files named.

-*l* Only list the names of files that contain a match to the expression as specified in -*e*.

-*p* **package** Restrict the search to a particular package (see §1.1.1, p1). The default is to search all packages.

Whitespace is allowed between short-form options and their arguments.

Diagnostics

Status return values
`0`: success
`1`: initialization error

Examples

The following command will search for all files in and below the `implement` directory of the `aips` package (that is `code/aips/implement/...`) whose name contains the string "aips" (case insensitive) and will search inside each file that it finds looking for the string "debug" (case insensitive).

```
squiz -c imp -e debug -p aips "*aips*"
```

The following simply counts the number of files in the `doc` package:

```
squiz -p doc | wc -l
```

Finally, this example will list out all files that contain the expression "plonk":

```
squiz -l -e plonk
```

See also

The unix manual page for *grep*(1).
§1.1.1, p1, AIPS++ directory structure.

Author

Original: 1994/07/28 by Mark Calabretta, ATNF
Modified: 1997/05/30 by Pat Murphy, NRAO

9.23 *tract*

Report the age of a file or directory.

Synopsis

tract [-*a*|-*c*|-*m*] [-*q*#] [-*s*] <file|directory>

Description

tract reports the age of a file or directory; by default it reports the time in seconds since the file or directory was last modified. In query mode it can be used to determine if a file or directory is older than a specified number of seconds.

Options

-*a* Report time since last access.

-*c* Report time since last status change.

-*m* Report time since last modification (default).

-*q*# Return a successful exit status if the file is older than the specified number of seconds. Nothing is reported on `stdout`.

-*s* Report the time in sexagesimal (h:m:s) format (default is seconds).

Diagnostics

Status return values
- `-1`: In query mode, the file is younger than the specified timespan.
- `0`: Success.
- `1`: Usage error.
- `2`: File access error.

Examples

The following reports the elapsed time in h:m:s format since tract was installed:

```
tract -s `which tract`
```

See also

The unix manual page for *find*(1).

Author

Original: 1995/02/21 by Mark Calabretta, ATNF

9.24 *xrcs*

Verify the internal consistency of RCS version files.

Synopsis

xrcs [-*q*] [file(s) | directory]

Description

xrcs exercises RCS version files. If a list of files is specified then *xrcs* will verify each; if a directory then all RCS version files in it and in all subdirectories will be verified. Files and directories may not be mixed.

File names may be specified without the `,v` suffix and without an `RCS` subdirectory specification. If `<file>` is the filename the search order is:

```
<file>,v
RCS/<file>,v
<file>
RCS/<file>
```

xrcs verifies each version file simply by checking out the lowest revision. Since this requires *co* to reconstruct every revision (on the trunk) it can only succeed if the whole version file is synactically correct.

If no file or directory is specified then `$AIPSROOT/slave` is assumed.

Options

-*q* Quiet mode, if a list of files was specified then don't report a successful verification. If a directory was specified, don't report the name and revision of each file in a directory as it is processed. A summary of any bad files found is always produced.

Notes

- *xrcs* is primarily used by *exhale* (p73) when producing a new base release.

Diagnostics

Status return values
0: success
1: initialization error
2: a bad RCS version file was found

Examples

To verify the RCS version file for *xrcs* itself:

```
cd ~/aips++/code/install/codemgmt
xrcs xrcs
```

To verify the AIPS++ slave repository in its entirety:

```
xrcs
```

See also

The unix manual page for *co* (1).
The unix manual page for *rcs* (1).
AIPS++ variable names (§1.2, p8).
exhale (p73), AIPS++ code export utility.

Author

Original: 1994/08/10 by Mark Calabretta, ATNF.

Chapter 10

System generation

Utilities for installing AIPS++ and compiling object libraries, executables, and documentation [1].

10.1 *configure*

AIPS++ installation script.

Synopsis

configure

Description

configure is used as the first step in installing AIPS++, either from scratch, or to add a new architecture (see §2.2, p15) and (§2.3, p15). In the latter case the installation of the *aipsinit* (p31) scripts and databases can be skipped.

A copy of *configure* resides in `aips2.nrao.edu:/pub/aips++/code_ftp` on the AIPS++ anonymous *ftp* server together with the compressed *tar* files which constitute an AIPS++ installation kit (see §1.1.5, p7 for a description of the AIPS++ ftp directory structure).

Once *configure* has been activated the installation consists of answering a few questions, most of which have sensible default answers. *configure* first ensures that certain utilities that it needs are available. These are GNU *gzip* and GNU *make* (p154). If they aren't it fetches them from the consortium site and if necessary, installs *gzip* temporarily in the current directory. It then unloads any compressed *tar* files present in the current directory.

configure next installs the *aipsinit* (p31) scripts and constructs the `$AIPSROOT/aipshosts` (p35) database via a simple inbuilt editor (AIPS++ variable names are listed in §1.2, p8). Once the host architectures, site-name, and host-names are known, *configure* invokes *aipsinit* for itself and then creates the AIPS++ system directory tree for each architecture. With a `bin` directory now in existence, *configure* reinstalls *gzip* if necessary, and likewise *gmake* (p154).

[1] Last change: Id: SysGen.tex,v 19.0 2003/07/16 04:18:25 aips2adm Exp

Before *gmake* can be invoked, however, the site-specific `makedefs` (p155) file must be defined. This contains definitions used by *gmake* for directories, compiler options and the like. Generic definitions are provided in `$AIPSARCH/makedefs` but several of these have no sensible default and must be defined on a per-installation basis. If none already exists, *configure* installs a template `makedefs` (p155) file in `$AIPSSITE` and invites the installer to make the appropriate entries with a text editor in another window (or by suspending *configure*). When the user is satisfied, *configure* checks to make sure that all definitions are sensible (see `testdefs` (p258)), and then repeats the process for the site-specific `aipsrc` (p38)).

At this point the system should have been bootstrapped to a state where a recursive *gmake* may be used to check-out all code and build the system, and this *configure* now does (see §10, p149).

Options

None.

Resources

Construction of the `aipsrc` databases is part of *configure*'s job.

Diagnostics

Status return values correspond to the various phases of the operation

- 0: success
- 1: initialization error
- 2: error fetching required utilities or installing *gzip*
- 3: error extracting compressed *tar* files
- 4: error installing the *aipsinit* (p31) scripts or databases
- 5: error constructing the system directory tree
- 6: error installing *gzip*, *gmake*
- 7: error installing local `makedefs` and `aipsinit` files

See also

The GNU manual page for *gzip*.
The GNU manual page for *gmake*.
AIPS++ variable names (§1.2, p8).
Installing an AIPS++ production-line system (§2.2, p15).
Installing an AIPS++ code-development system (§2.3, p15).
`aipshosts` (p35), AIPS++ hosts database.
`aipsrc` (p38), AIPS++ resource database.
getrc (p43), query AIPS++ resource database.
`makedefs` (p155), AIPS++ makefile definitions.
`testdefs` (p258), test `makedefs` (p155) definitions.

Author

Original: 1992/04/09 by Mark Calabretta, ATNF.

10.2 *depend*

Produce a dependency list for AIPS++ class implementation files.

Synopsis

depend *-p* `package file1 [file2...]`

Description

depend produces a list of dependencies of AIPS++ class implementation files in relation to their included header files. The dependency lists and timestamp files for each package/module are stored in `$ARCHAUXD` (AIPS++ system directories are described in §1.1.3, p4). The depencency lists have names of the form ***package-module***`.list`. The module name may be empty to indicate class implementation files which reside in the `implement` directory itself.

depend works for system compilations only, dependency analysis for programmer compilations is handled separately.

The `implement makefile` (p162) includes the dependency list associated with the package and module of the directory that it is processing. Thus, ***depend*** is normally invoked automatically by the `implement` makefile as part of the process of updating the files included by it. The list of files passed to ***depend*** in that case consists of all `.cc` files for the module and all `.h` files newer than the `.time` timestamp file produced by ***depend*** on its last execution.

Beware that invoking ***depend*** manually with an incomplete list of files could invalidate the dependency analysis.

The C++ header and implementation files may be specified in full or in brief. For example, in full:

```
/aips++/code/aips/implement/Foo.cc
/aips++/code/aips/implement/Foo.h
/aips++/code/include/aips/Foo.h

/aips++/code/aips/implement/module/Bar.cc
/aips++/code/aips/implement/module/Bar.h
/aips++/code/include/aips/module/Bar.h
```

(the last two are the same file) or in brief:

```
Foo.cc
aips/Foo.h

module/Bar.cc
aips/module/Bar.h
```

Note that the implementation files must correspond to the package-module combination specified via the ***-p*** option, but the header files may belong to any package and so must have a package and module name associated with them.

Options

-p `package[-module]` The AIPS++ package-module combination. There is no default for the package name. An empty module name indicates the `implement` directory itself.

Notes

- The complete list of header and implementation files for a package given in full form may blow the environment in some operating systems and the brief form should therefore be used.
- ***depend*** is designed for parallel execution on any number of machines. It uses a "pull-queue" mechanism so that no machine needs to know about any of the others. Parallel execution may be implemented for installations which maintain more than one architecture by defining `$ARCHAUXD` for the second and subsequent architectures to be symbolic links to `$ARCHAUXD` for the first architecture.
- The dependency analysis done for programmer compilations does not use ***depend***.

Diagnostics

Status return values
`0`: success
`1`: initialization error

Examples

depend should not be invoked manually (as explained above). It is used by the `implement makefile` (p162).

Files

`$ARCHAUXD/`*`package-module`*`.list` ...the dependency list for a package and module.

`$ARCHAUXD/`*`package-module`*`.time` ...timestamp file for the dependency list.

See also

AIPS++ system directories (§1.1.3, p4).
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

Author

Original: 1993/01/25 by Mark Calabretta, ATNF

10.3 *gmake*

GNU *make* - a program for directing recompilation.

Synopsis

`gmake [options] [targets]`

Description

gmake is the name used within AIPS++ to differentiate the GNU*make* executable from the many other flavours of *make* available on unix systems.

GNU *make* is essential for AIPS++, and is installed automatically by *configure* (p149) as "*gmake*" when AIPS++ is first installed.

GNU *make* has many features which make it preferable to any other form of *make*. It can process makefiles written for most flavours of *make* although the converse is not true.

GNU *make* is free software provided by the Free Software Foundation.

Notes

- Version 3.71 or later is required.
- POSIX.2 compliance for *make* applies largely to the way makefiles themselves are written, and essentially restricts functionality to the lowest common denominator of all the many varieties of *make*. This means it would be it very difficult to write portable POSIX.2 compliant makefiles for AIPS++, so GNU `Make` has been adopted instead. GNU `Make` is itself POSIX.1 and POSIX.2 compliant, and so satisfies the "POSIX compliance" criteria originally set for AIPS++.

See also

The GNU `Make` manual.
The GNU manual page for *make*.
configure (p149), the AIPS++ installation script.
`makedefs` (p155), AIPS++ makefile definitions.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

10.4 makedefs

Installation-specific definitions used by the AIPS++ makefiles.

Synopsis

`makedefs`

Description

The **makedefs** file is a GNU makefile which defines variables and rules common to all AIPS++ makefiles. It contains comments addressed to AIPS++managers who must construct and maintain a site-specific version and as such is intended to be self-documenting. However, the following notes are provided as a general overview.

Default definitions

The AIPS++ **makedefs** files contain definitions used by the AIPS++ makefiles for object libraries, include directories and paths, compiler options and the like.

Every AIPS++ makefile includes the default **makedefs** file at an early stage via a sequence similar to the following:

```
# AIPS++ fundamentals.
#---------------------
AIPSROOT := $(word 1, $(AIPSPATH))
AIPSARCH := $(AIPSROOT)/$(word 2, $(AIPSPATH))
include $(AIPSARCH)/makedefs
```

(see §1.2, p8 for AIPS++ variable names).

The default **makedefs** guarantees to supply a definition for every primary variable used by the AIPS++ makefiles to control compilations. While the default values supplied for some of these are unlikely to be correct, they are provided as an indication of what may need to be redefined in the site-specific **makedefs** file for each architecture.

The default **makedefs** file also defines many other variables, such as directory locations, which are applicable to all AIPS++ installations and do not need to be redefined.

Site-specific definitions

While **$AIPSARCH/makedefs** contains generic definitions of variables used in the AIPS++ makefiles, a number of these have no sensible defaults and must be redefined on a site-wide basis for each architecture in **$AIPSSITE/makedefs** (§1.1.3, p4 describes the AIPS++ system directory structure). The default file includes the site-specific file and definitions made in the latter override those of the former.

Part of the process of installing AIPS++ involves defining the site-specific `makedefs` (p155) file. The installation utility, ***configure*** (p149), provides a template version tailored for each architecture and requires the installer to carefully read through the default `makedefs` file and make the appropriate redefinitions. ***configure*** then uses `testdefs` (p258) to ensure that all of the definitions are sensible.

As an example, the default `makedefs` file contains the following definitions for the location of the X11 windowing system's include files and libraries:

```
X11ROOT  := /usr/local/X11
X11INCD   = $(X11ROOT)/include
X11LIB    = -L$(X11ROOT)/lib -lX11
```

Note that `X11ROOT` is defined as a "simply expanded" variable in GNU*make*. These are defined using "`:=`" and are evaluated immediately. On the other hand, `X11INCD` and `X11LIB` are defined as "recursively expanded" variables expressed in terms of `X11ROOT`. Recursively expanded variables are defined using "`=`" and are evaluated when used. This construct allows the site-specific `makedefs` files to redefine `X11ROOT` without necessarily needing to redefine the other two.

As an example, supposing that X11 was installed locally in `/usr/X11` with include files and libraries in the usual `include` and `lib` subdirectories, then the site-specific `makedefs` need only make the following redefinition:

```
X11ROOT  := /usr/X11
```

Note that all variable definitions in the site-specific `makedefs` files should be "simply expanded". In general the site-specific files should not redefine any more than is necessary; this is particularly the case for AIPS++ consortium installations which are continually evolving. For example, if at some later time an AIPS++ application was written which used the Athena widget set, then the default value of `X11LIB` might be reset to

```
X11LIB    = -L$(X11ROOT)/lib -lXaw -lX11
```

and in the above example the change to the default value of `X11LIB` would have effect without any changes required to the site-specific `makedefs`. On the other hand, if the site-specific `makedefs` file had redefined `X11LIB` then it would have to be amended to add the `Xaw` library.

System versus programmer usage

As described in the entry for `makefiles` (p162), the AIPS++ makefiles serve the dual purpose of directing both system and programmer compilations. These programmer-oriented definitions are only of interest in AIPS++ development systems and may safely be ignored in production line installations.

System and programmer oriented variables and rules mostly reside together in the makefiles without conflict. To help resolve the few cases where conflicts can occur `makedefs` sets a variable, `MAKEMODE`, to either "`system`" or "`programmer`" depending on whether the *gmake* (p154) was initiated from a system or programmer directory. It is currently used to choose the default target and to circumvent certain time consuming system dependency analyses which are not required for programmer compilations.

Other variables defined within `makedefs`

The default `makedefs` defines a collection of variables which are not permitted to be redefined in the site-specific `makedefs`. Of these, one group may be redefined on the ***gmake*** (p154) command line. These mainly define the location of AIPS++ directories and although there are rare occasions when the AIPS++ code distribution system needs to reset them, this should never be necessary in normal usage.

A second small group of "immutable" variables defined at the start of `makedefs` are used mainly for job control and syntactic purposes in the AIPS++ makefiles. These are not permitted to be redefined even on the ***gmake*** command line.

Rules defined in `makedefs`

A cursory glance through `$AIPSARCH/makedefs` will be sufficient to show that it contains much more than just default variable definitions. In fact, it is a self-contained GNU makefile and can be, and is, invoked by itself by some parts of the AIPS++ build system. However, these "nether" sections of `makedefs` can safely be ignored by AIPS++ system managers.

Since it is included by all AIPS++ makefiles, `makedefs` is the appropriate place to define the many rules which are common to all of them. It would be very difficult if not impossible to understand the functioning of any of the AIPS++ makefiles without reference to `makedefs`.

The main pattern rule defined within `makedefs` is that for checking out source code (if the AIPS++ installation includes RCS repositories). Apart from that, `makedefs` contains many static rules; their definitions fall into several categories (refer to the GNU ***make*** manual for an explanation of how rules are written):

- Single-colon rules without commands, with or without dependencies; an including makefile may define the commands and add additional dependencies. Examples are `all` and `allsys`, `docsys`, `docscan` and `docextr`.

  `FORCE`, the target of a single-colon rule with no dependencies or commands, is used as a dependency to force a target to be remade. It should not be altered by an including makefile.

- Single-colon rules with commands, with or without dependencies; an including makefile may add additional dependencies but no additional commands. These rules are usually meant to be completely defined within `makedefs` and not augmented by the including makefile. Examples are `chkout`, `eval_vars` and `show_sys`.

  Some of this class of rules such as `dependsys` are defined in `makedefs` solely to establish their recursive nature. Such rules may be invoked in any directory and will recurse into every subdirectory but only actually do anything in the directories where the rule is augmented with a dependency list by the makefile which includes `makedefs`.

  Others of this class of rules have as a dependency a phony target of the same name preceded by a "`.`" which may be defined in the including makefile to do specific operations *before* the commands defined in the `makedefs` rule are executed. These phony targets are also declared in `makedefs` to save having to do that in the including makefile if they're not required. The `clean` target is an example of this.

- Double-colon rules with commands, with or without dependencies; an including makefile may define double-colon rules of the same name with commands, with or without dependencies; these rules are independent and are executed sequentially. The `help` target is an example, the `makedefs` rule for this lists general targets and the corresponding rule in the including makefile lists additional targets defined therein.

Organized by category, `makedefs` (p155) declares and possibly lists dependencies and defines rules for the following targets (see also `makefiles` (p162)):

- `.PHONY` target declarations.
- Declaration of the default target `allsys` for system invokations and `all` for programmer invokations.
- General purpose targets such as `FORCE` and `null`.
- Rules for `makedefs` and the makefiles to update themselves.
- The `chkout` target for recursively checking out all sources (if the RCS repositories are present).
- Rules for processing subdirectories by name, both for system and programmer.
- Rules for creating the AIPS++ system directories if they don't already exist.
- General `clean` type rules for deleting intermediate files. These have as a dependency a target whose rule may be supplied by the including makefile for deleting intermediate files which are specific to that makefile.
- A rule, `recurse`, which recursively invokes a non-recursive rule specified by the value of the `TARGET` variable. For example, `gmake TARGET=clean recurse`.
- A rule, `command`, which invokes a command specified by the value of the `COMMAND` variable. For example, `gmake COMMAND=env command`.
- Rules defined for diagnostic purposes, including rules for reporting variable definitions.
- A `help` target which lists the main targets defined in `makedefs`. This is defined as a double-colon rule and the including makefile can supply the corresponding rule to list additional targets defined therein.

The makefile targets are discussed in detail elsewhere (§10.6, p162).

Notes

- The site-specific `makedefs` is a valid makefile in its own right and in principle new features could be added through it, in particular, new targets and conventions. In practice, such features should be added judiciously lest the AIPS++ programming environment deviate too far from the norm and make it difficult to diagnose problems.

  Useful features may be developed and evaluated in the site `makedefs` before being incorporated into the default `makedefs`.

- The default `makedefs` is a critical part of the AIPS++ code distribution system. If a syntactically incorrect version was distributed to consortium sites, and if the damage was such that the broken version could not install a repaired version subsequently distributed, then the rebuild part of the code distribution system would be broken.

Files

`$AIPSARCH/makedefs` ...defaults values.

`$AIPSSITE/makedefs` ...site-specific values for each architecture.

See also

The *GNU make* manual.
The GNU manual page for ***gmake***.
AIPS++ variable names (§1.2, p8).
configure (p149), the AIPS++ installation script.
gmake (p154), GNU make.
`makedefs adjuncts` (p160), Per-sourcefile makedefs definitions.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.
`testdefs` (p258), test `makedefs` definitions.

Author

Original: 1992/05/01 by Mark Calabretta, ATNF

10.5 `makedefs` adjuncts

Per-sourcefile makedefs definitions.

Description

The `app`, `imp`, `ftn` and `tst` makefiles (§10.6, p162) look for and, if found, include an adjunct `makedefs` (p155) file which may contain definitions (or, in principle, rules) which apply to a particular source code directory. The adjunct files are architecture-specific and are primarily intended to control compiler options on a per-sourcefile basis.

The adjunct files reside in `$INSTARCH` (see §1.2, p8) and have names of the following form

```
                  Form                              Examples
         ---------------------------------   ---------------------
imp      <package>[-<module>].defs           aips.defs
                                             aips-Tables.defs
tst      <package>[-<module>]-test.defs      aips-Tables-test.defs
ftn      <package>-fortran[-<module>].defs   aips-fortran.defs
app      <package>_<app>.defs                aips_gtable.defs
```

As a simple example of the use of a `makedefs` adjunct, the applications makefile uses a variable called `BIN` as a hook to allow conditional compilation of an application. If `BIN` is set to blank in an adjunct file for a particular architecture then compilation of that particular application will be skipped during a recursive system build.

The following adjunct `makedefs` decreases the optimization level for `TSMCube.cc` for the GNU C++ compiler running under OSF/1. The file, `$(AIPSCODE)/install/alpha_gnu/aips-Tables.defs`, is included by the implement makefile, and it illustrates the general scheme:

```
# Switch off optimization for TSMCube.

LVL  := $(C++OPTLVL)
LVL0 := -O0

TSMCube := 0

C++OPTLVL = $(LVL$($*))
```

Although this is a non-trivial construct the general form should be easy to follow. The operative part is the last line in which `C++OPTLVL`, a recursively expanded `makedefs` variable which determines the C++optimization level, is evaluated as a "computed" variable within the pattern rule which compiles `TSMCube.o` from `TSMCube.cc`:

1. Within the pattern rule, `$*` is an "automatic" variable which evaluates to the stem with which the pattern rule matches, in this case `TSMCube`. After the first round of translation `C++OPTLVL` would be set to `$(LVL$(TSMCube))`.

2. `$(TSMCube)` is evaluated in the second round of translation after which `C++OPTLVL` would become `$(LVL0)`.

3. The required result, `C++OPTLVL = -O0`, is achieved after the third round of translation.

For any other source file in the `$(AIPSCODE)/aips/implement/Tables/` directory, say `Table.cc`, the sequence would be as follows:

1. After the first round of translation `C++OPTLVL` would be set to `$(LVL$(Table))`.

2. In the second round of translation `$(Table)` is undefined and evaluates to an empty string whence `C++OPTLVL` would become `$(LVL)`.

3. After the third round of translation `C++OPTLVL` is set to its original value by virtue of the simply expanded variable assignment of `LVL` at the start of the adjunct makedefs.

The default case relied on `Table` being undefined as a makefile variable. This is a safe assumption since makefile variable names are case-sensitive and while AIPS++ source file names contain a mixture of upper- and lower-case characters, the `makedefs` and `makefile` variables in AIPS++ are all upper-case.

The `makedefs` variables which specify debug or optimization flags for the various compilers are

`C++DBGLVL` ...C++ debug flags

`C++OPTLVL` ...C++ optimization flags

`CDBGLVL` ...C debug flags

`COPTLVL` ...C optimization flags

`FCDBGLVL` ...FORTRAN debug flags

`FCOPTLVL` ...FORTRAN optimization flags

These variables are not used directly in the makefiles but are instead used as components in the definition of variables which contain all of the compiler flags. An imporant prerequisite for this system is that these variables, namely `C++DBG`, `C++OPT`, `CDBG`, `COPT`, `FCDBG`, and `FCOPT`, be recursively expanded. This also applies for the programmer compilation option variables `C++FLAGS`, `CFLAGS`, and `FCFLAGS`.

See also

The *GNU make* manual.
The GNU manual page for ***gmake***.
AIPS++ variable names (§1.2, p8).
gmake (p154), GNU make.
`makedefs` (p155), AIPS++ makefile definitions.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

10.6 `makefiles`

GNU makefiles used to rebuild AIPS++

Synopsis

```
makefile
makefile.{app,aps,chk,doc,ftn,imp,pkg,scr,tst}
```

Description

Compilation of the AIPS++ system from its sources is accomplished via a hierarchical system of GNU makefiles (§10.3, p154). The top-level makefile resides in `$AIPSCODE` (§1.2, p8) and certain targets cause all makefiles residing in the subdirectories below to be invoked recursively. In this way all of the AIPS++ executable scripts, object libraries, applications, and documentation may be compiled and installed with a single invokation of ***gmake***.

In addition to these *system-oriented* targets, the makefiles also have a set of *programmer-oriented* targets designed for code development.

Before examining the makefile targets in detail I will briefly summarize the general features of the AIPS++ system of makefiles:

- Every AIPS++ source directory contains a makefile called `makefile` (`.dir` subdirectories are the only valid exceptions). Most of these makefiles simply `include` one of the following *generic* makefiles from `$CODEINSD`:
  - `makefile.app` (application subdirectory)
  - `makefile.aps` (`apps` directory)
  - `makefile.chk` (checkout)
  - `makefile.doc` (documentation)
  - `makefile.ftn` (`fortran` directory)
  - `makefile.imp` (`implement` directory)
  - `makefile.pkg` (package)
  - `makefile.scr` (`scripts` directory)
  - `makefile.tst` (`test` directory)

  These are referred to below by their three-letter file suffix, namely `app`, `aps`, `chk`, `doc`, `ftn`, `imp`, `pkg`, `scr`, and `tst`.
- Every makefile includes (whether directly or indirectly) an architecture-specific makefile, `$AIPSARCH/makedefs` (p155), which defines variables and rules common to all AIPS++ makefiles.
- The makefiles contain a set of system-oriented targets for rebuilding the AIPS++ system (see §1.1.3, p4), and a set of programmer-oriented targets for code development in programmer

workspaces (§3.1.2, p20). They also contain diagnostic targets for reporting makefile variable definitions, versions of installed software, for running system verification procedures, and printing a list of targets and their functions.

- The makefiles recognize that the AIPS++ installation may or may not have a copy of the RCS source code repositories (see §1.1.4, p5). If the repositories are present, the plain-text copies are automatically updated if necessary. In particular, the makefiles contain targets which cause *gmake* (p154) to update the makefile itself and any included files before attempting to remake anything else.
- The makefiles usually do not contain lists of source files but instead construct them via GNU *make*'s `wildcard` function. This means that new files checked into AIPS++ are discovered automatically by the makefiles.
- Any subdirectory which contains a makefile is also recognized as a target.

A few directories contain special-purpose makefiles, the main ones being the top-level makefile in `$AIPSCODE` (referred to as `top` below), and the installation makefile in `$CODEINSD` (referred to below as `ins`).

The generic and special-purpose makefiles are collectively referred to below as *specific* makefiles, as opposed to the general makefile, `makedefs`.

File suffixes recognized by the `makefiles`

The generic makefiles recognize the following source file types:

`app`

`.cc`	... C++ implementation file.
`.h`	... C++ header file.
`.g`	... Glish script.
`templates`	... Template list (see *mkinst* (p180)).
`MyTemplatesOnly`	... Template instantiation semaphore file.

`aps`

none

`chk`

any

`doc`

`.latex`	... LaTeX file.
`.tex`	... TeX file.
`.texi`	... Texinfo file.
`.ps`	... PostScript file.
`.text`	... ASCII text file.
`.txt`	... ASCII text file.
`.html`	... HTML file.
`.htm`	... HTML file.

`.gif`	... GIF file.
`.jpg`	... JPEG file.
`.dir`	... Subdirectory containing files included by the document.

`ftn`

`.f`	... FORTRAN source file.

`imp`

`.cc`	... C++ implementation file.
`.h`	... C++ header file.
`.l`	... *lex* source file.
`.y`	... *yacc* source file.
`.g`	... Glish script.
`.gif`	... GIF file.
`.html`	... HTML file.
`templates`	... Template list (see *mkinst* (p180)).
`MyTemplatesOnly`	... Template instantiation semaphore file.

`pkg`

none

`scr`

any

`tst`

`.cc`	... C++ implementation file.
`.h`	... C++ header file.
`.g`	... Glish script.
`templates`	... Template list (see *mkinst* (p180)).
`MyTemplatesOnly`	... Template instantiation semaphore file.

Section §10.9, p171 describes the makefile rules used to process these source files and the type of files produced.

10.7 Object libraries

Generation of AIPS++ object libraries and usage of external libraries in AIPS++.

Library control variables

An AIPS++ installation may be configured to maintain a debug version of the object libraries, an optimized version, or both as determined by the setting of these variables. They have names of the form `LIB`<*package*>, for example `LIBaips`, and their values consist of six fields corresponding to the debug and optimized C++, `C`, and FORTRAN libraries in that order,

```
#                     C++           C          FORTRAN
#                  dbg    opt    dbg    opt    dbg    opt
#                 ------------- ------------- -------------
LIBaips        := static static defeat defeat static static
```

Four keywords are recognized:

- **defeat**: The library will not be built (and if an old version exists it will be deleted).
- **static**: The static library is built and is the one linked to.
- **shatic**: The static and shared libraries are built but the static library is the one linked to.
- **shared**: The static and shared libraries are built and the shared library is the one linked to.

The library control variables are used by the AIPS++ `makefiles` (p162) to determine whether to compile the debug and/or optimized versions of an object module, and which libraries an executable may be linked with. The library updating procedure, ***updatelib*** (p183), also uses them to determine which sharable libraries to build.

External libraries

AIPS++ uses many external libraries such as `PGPLOT`, `LAPACK`, and `X11`. These are expected to be installed separately from AIPS++. Other libraries such as `Glish` and `WCSLIB` while bundled with AIPS++ are nevertheless treated by `makedefs` (p155) as though they were external.

`makedefs` (p155) contains a set of variables for each external library. For example, for `Tcl` the variables are

```
TCLROOT  := /usr/local
TCLDEFS  := -DTCL
TCLINCD   = $(TCLROOT)/include
TCLLIBD   = $(TCLROOT)/lib
TCLLIB   := -ltcl
```

The corresponding variables for `Glish` and `WCSLIB` are set to point to the AIPS++ system directories where they are installed.

The per-directory makefile for a class implementation module, an application or test program registers its usage of one or more external libraries via the `XLIBLIST` variable. For example,

```
XLIBLIST := GLISH WCSLIB F77
include $(word 1, $(AIPSPATH))/code/install/makefile.app
```

This examples tells `makedefs` (p155) to construct compile and link options using `Glish`, `WCSLIB`, and `FORTRAN`. Note that the order is important, `WCSLIB` comes *before* `F77` in the list since it uses `F77` and not vice versa. It is valid for a library to be repeated although that should not be necessary.

Note also that the mnemonics used for the external libraries are case-sensitive. Most are all-uppercase, `Motif` is the one exception.

The `*ROOT` variables in the external library definitions are not used directly by `makedefs` (p155) but may allow the site-specific `makedefs` to provide a simple redefinition as explained elsewhere (§10.4, p155). The other four variables are only used within `makedefs` as follows:

- The `*DEFS` variables are collected into a variable called `XTRNDEFS` which is appended to the C-preprocessor options (`CPPDBG` and `CPPOPT`) in `makedefs` (p155). `XTRNDEFS` is not used outside `makedefs`.
- The `*INCD` variables are given a `-I` prefix and collected together after `$(AIPSCODE)/include` in a variable called `AIPSINCL` which is used as a compiler option in the generic makefiles.
- The `*LIBD` variables are collected into a variable called `XLIBPATH` which is only used within `makedefs` (p155) to construct `XTRNLIBS` (see next item).
- The `*LIB` variables are appended to `$(XLIBPATH)` and there followed by `$(ARCHLIBS)` (platform-specific unix system libraries) to form `XTRNLIBS`. `XTRNLIBS` is used in the link list by the `app` and `tst` makefiles.

While external libraries may have preprocessor definitions associated with them, it is also possible to add preprocessor directives to a per-directory makefile via the `CPPMODU` variable. `CPPMODU` is processed within `makedefs` the same way as preprocessor definitions defined for each external library, that is, they are appended to the C-preprocessor options `CPPDBG` and `CPPOPT` and not otherwise used outside `makedefs`. An example of its usage

```
XLIBLIST := GLISH WCSLIB F77
CPPMODU  := -DAIPS_UNAME
include $(word 1, $(AIPSPATH))/code/install/makefile.app
```

Note that preprocessor macro definitions made via `CPPMODU` must not be platform-specific since this would impede portability. Any such definitions should be made in `makedefs`.

See also

`makedefs` (p155), AIPS++ makefile definitions.
`testdefs` (p258), test `makedefs` definitions.
updatelib (p183), update an AIPS++ object library.

10.8 Template instantiation

C++ template instantiation in AIPS++

C++ template classes

AIPS++ makes heavy use of C++ template classes, also referred to as "parameterized types". These are typically used to implement "container" classes, for example `Set`, `List`, and `Array` classes, which serve to aggregate objects of another type. In C++, `Set` would be defined in "template" form as `Set<T>`, where the template type (or parameter), `T`, represents some as yet unspecified data type. This saves having to code `Set` classes separately for `int`, `float`, or general `Object` data types.

The C++ compiler "instantiates" the `Set<T>` class template for objects of type `float` (say) when it compiles a piece of code which declares that it wants to use `Set<float>`.

Global functions can also be templated. The argument list and return value may be defined in terms of template type `T`, where `T` often (but not necessarily) refers to a template class.

It often happens that particular template classes and functions can be implemented more efficiently for a particular template type by manual coding rather than by instantiation of the class or function template. C++allows "template specializations" to be written for particular instances of template classes, but some compilers may require these to be registered in a particular way.

Instantiation mechanisms

While no two C++ compilers use exactly the same template instantiation mechanism there are two basic types:

automatic: Some compilers deduce what templates need to be instantiated, the only outside intervention needed being provided by compile-line options. These compilers maintain a "template repository" which holds intermediate files generated in the instantiation process. Older CFRONT-based compilers did all template instantiation at link time, but more modern "native" compilers do as much as possible when the object library is generated. They include template object files together with those generated from the class implementation files in order to speed up the linking phase.

Since the compiler needs to locate all C++ class implementation and header files required for the instantiation this is a non-trivial operation and this method is notoriously bug-ridden.

manual: The GNU C++ compiler, *g++*, cannot deduce what templates need to be instantiated. However, it does accept a small C++ source file which includes the necessary class implementation and header files and declares the template of the desired type.

The proliferation of template instantiation mechanisms has presented a significant challenge for the AIPS++ system, particularly in avoiding the introduction of compiler-specific procedures.

Template repositories

Compilers which maintain template repositories have mostly been accomodated via the use of PTRDBGD, PTROPTD variables in makedefs, and also in distinguishing between the archiver used for C++ object libraries (CXXAR) and other libraries (AR).

Template specialization and the Sun native compiler

The Sun native compiler requires template specializations to be registered in the template instantiation options file, Template.opt, which resides in the template repository. Within AIPS++ these specializations are registered in specializations files which reside in the same directory as the C++ code. A specialize rule is supplied with the initial version of makedefs supplied for sun4sol_ntv installations, and specialize is added as a dependency of allsys. The specialize rule checks for the existence of a specializations file and if found merges its contents into Template.opt. In this way the compiler specific behaviour is isolated to the local makedefs file and a set of specializations files which are ignored by other compilers.

Do-it-yourself (diy) templates

The GNU C++ compiler, *g++*, must have template instantiation files supplied to it. This "do-it-yourself" mode is enabled by setting the TEMPLATES variable in makedefs (p155) to diy.

Templates to be instantiated are discovered during code development via the list of unresolved symbols which results when attempting to compile a program. The linker reports the mangled names of the templates and these may be demangled with the GNU version of *c++filt* (the name mangling scheme used by *g++* differs from other compilers).

It is up to the programmer to supply the name of the class implementation file and any required header files for each template to be instantiated. This information is recorded as an entry in the templates file as described in the entry for *mkinst* (p180). The templates files are maintained within the AIPS++ system and programmers may also create private copies for code development.

templates files may reside in any imp, app or tst directory. However, these are handled in different ways when compiling the system (see chapter 11 for programmer usage):

- Template instantiations generated in an implement directory are added to the related system object library so that they may be shared by all applications and test programs which use that library. By convention such templates files are restricted to the _ReposFiller "module" for each AIPS++ package.
- app directories contain only a single application so the template instantiation files generated therein are not pre-compiled but instead are added to the compilation command for the application.
- Since tst directories may contain several test programs the template instantiation files are pre-compiled and inserted into a temporary object library which is deleted after all test programs have been compiled.

See also

mkinst (p180), Generate template instantiation files from a list.
reident (p214), Remake the idents in a `templates` list.
duplicates (p216), Checks for duplicate entries in `templates` lists across the system.

10.9 System generation makefile rules

GNU makefile rules used to rebuild AIPS++

Synopsis

```
makefile
makefile.{app,aps,chk,doc,ftn,imp,pkg,scr,tst}
```

AIPS++ targets

The AIPS++ makefile targets are listed below by category. These lists are not exhaustive, but do aim to cover everything of practical use. In particular, they omit targets which are intended for the internal use of the makefiles.

A target is labelled as "recursive" if it causes ***gmake*** (p154) to be invoked in all subdirectories. It is "general" if it applies to all makefiles; such targets are defined in `makedefs`. A target is "specific" if defined in a specific makefile.

Some targets such as `allsys` have a general meaning, the specific behaviour of which differs for specific makefiles. These are referred to as "general/specific" and where appropriate the details of a target's behaviour are described for each of the generic makefiles, for the top-level makefile (`top`), and the installation makefile (`ins`).

Targets which apply only if the RCS source code repositories are present are marked as "RCS".

Rules for system-oriented targets always take source code from the `$AIPSCODE` subdirectories. If `$AIPSRCS` exists, then these plain-text sources are updated from the RCS repository first if necessary.

- `allsys` : (default system target, general/specific, recursive)
  This is declared generally in `makedefs` as the default target if ***gmake*** has been invoked from a subdirectory of `$AIPSCODE`. It has a dependency which causes source files to be checked out where required if the RCS repositories exist. Further dependencies and commands are defined in the specific makefiles to compile and install all AIPS++ sources.
  - `app`: Rebuild the application; template instantiation files will be created from a `templates` file if necessary. Also install any Glish scripts in `$(LIBEXECD)`.
  - `aps`: Rebuild all applications residing in an `apps` directory.
  - `chk`: Just update the plain-text sources if RCS repositories are present.
  - `doc`: Compile all documentation.
  - `ftn`: Rebuild a FORTRAN library from its sources.
  - `imp`: Rebuild a C++ library from its sources updating the system dependency list in the process (see ***depend*** (p152)). Also install any Glish scripts in `$(LIBEXECD)`.
  - `ins`: Compile and install all AIPS++ system utilities in the `bin` area. See also `thisys`.
  - `pkg`: Rebuild a whole package. Auxiliary directories such as `glish` and `wcslib` in the `aips` package are done first, then the `scripts` subdirectory, then the AIPS++ libraries, then all applications.

- `scr`: Install all AIPS++ executable scripts in the `bin` area and install any Glish scripts in `$(LIBEXECD)`. When *perl* scripts are installed the pathname in the strings `#!/usr/local/bin/perl4` and `#!/usr/local/bin/perl5` on the first line of the script are replaced with the pathname encoded in the `PERL4` and `PERL5` `makedefs` variables. All sources present in the directory will be installed except for those listed in the `EXCLUDE` variable. By default, this contains the name of the makefile itself to prevent it from being installed.
- `top`: Rebuild the entire AIPS++ system as follows:

  If necessary, the `include` subdirectory is first created and populated with symbolic links to the `implement` directories of all installed packages (see *mktree* (p126)).

  Before commencing the rebuild any required system directories are created.

  The `install`, `aips`, `dish`, `synthesis`, and `vlbi` subdirectories are then built in that order.

  If the `admin` subdirectory exists it is then rebuilt.

  Auxiliary packages are then rebuilt under the control of the `AUXILIARY` variable. This may be set in `makedefs` to the required combination of `contrib` and `trial`. They will be rebuilt in the order listed in `AUXILIARY`.

  Consortium packages are then rebuilt under the control of the `CONSORTIA` variable. This may be set in `makedefs` to the required combination of `atnf`, `bima`, `hia`, `nfra`, `nral`, `nrao`, and `tifr`. They will be rebuilt in the order listed in `CONSORTIA`.

  The AIPS++ documentation is then compiled. This can be defeated by unsetting the `DOCSYS` variable in `makedefs`.

  Document extraction from C++ sources is then performed under the control of the `DOCEXTR` variable in `makedefs`.
- `tst`: Create template instantiation files from a `templates` file if necessary. If the `BINTEST` variable is set to "`bintest`" (or just "`bin`") in `makedefs` then rebuild all test programs. Debug compilations are done by default, but if the `TESTOPT` variable is set to "`opt`" in `makedefs` then optimized compilations will be done.

• `thisys` : (specific)
Non-recursive version of `allsys`.

- `ins`: Compile and install all AIPS++ system utilities in the `bin` area. Same as `allsys` except that it does not recurse into any subdirectories.

• `aipsroot` : (specific)
Special purpose rule to remake the `$(AIPSROOT)` directory.

- `ins`: Install all root level files.

• `$(SUBDIRS)` : (general, recursive)
`makedefs` sets the `SUBDIRS` variable to the names of all subdirectories which have a makefile, thus making them recognized targets. If the RCS source code repositories exist then `SUBDIRS` lists the subdirectories of the corresponding RCS directory. This list is then made the target of a rule for creating the subdirectory and, if the RCS directories are present, its `RCS` symlink (see *mktree* (p126)). A *gmake* is then initiated within the subdirectory. Note that this target has a programmer-oriented counterpart which behaves similarly.

- `$(CODESUBS)` : (general, recursive)
  Same as `$(SUBDIRS)` except that a subdirectory of `$AIPSCODE` may be specified via an absolute pathname.

- `null` : (general)
  Does nothing directly, but indirectly it may cause a makefile to update its included files, and in particular, this may trigger a dependency analysis to be done (see ***depend*** (p152)).

- `chkout` : (general, recursive, RCS)
  Update the `$AIPSCODE` directories by checking out files from the corresponding `$(AIPSRCS)` directories (if the RCS repositories are present).

  In installations which use do-it-yourself template instantiation (§10.8, p168) the `chkout` rule also generates template instantiation files in the `tmplinst` subdirectory by invoking ***mkinst*** (p180). (These `tmplinst` subdirectories are maintained by the makefile in their parent directory and should not contain their own makefile.)

  The `chkout` target for the `doc` makefile explicitly handles all subdirectories with names of the form `.dir` so that these subdirectories need not and should not contain a makefile.

- `symlinks` : (specific)
  Create the `$(CODEINCD)` directory and symlinks.

  - `top`: Create the `$(CODEINCD)` directory and symlinks.

- `sysdirs` : (general)
  Create the AIPS++ system directories, the subdirectories of `$AIPSARCH`. This is invoked by the top-level (`$AIPSCODE`) makefile at the beginning of a recursive rebuild to ensure that all required system directories exist.

- `cleancode` : (general/specific, recursive)
  Recursively delete files from the AIPS++ code areas which shouldn't be there - since noone should work from within these areas such files should not normally accumulate in them. Generally, the following are deleted: `,*,`, `*` , `*%`, `a.out`, and `core`. The general target has a dependency which may be defined in the specific makefiles as the target of a rule to delete additional files:

  - `app`: `*.i`, `*.o`, `*.cdb`, `*.cyi`, and any `tmplinst` subdirectory.
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: `*.aux`, `*.bbl`, `*.blg`, `*.cp`, `*.cps`, `*.dvi`, `*.fi`, `*.fis`, `*.fn`, `*.fns`, `*.ky`, `*.kys`, `*.lof`, `*.log`, `*.pg`, `*.pgs`, `*.toc`, `*.tp`, `*.tps`, `*.vr`, `*.vrs`, `*.info`, `*.info-*`
  - `ftn`: `*.o`
  - `imp`: `*.lcc`, `*.ycc`, `*.i`, `*.o`, `*.cdb`, `*.cyi`, and any `tmplinst` subdirectory.
  - `ins`: `*.i`, `*.o`
  - `pkg`: (none)
  - `scr`: (none)
  - `top`: (none)
  - `tst`: `*.i`, `*.o`, `*.cdb`, `*.cyi`, `*_tmp*` (temporaries created by ***assay*** (p249)), and any `tmplinst` subdirectory.

- `cleansys` : (general/specific, recursive)
  Delete selected files from the AIPS++ system directories, usually as a prelude to rebuilding the system in its entirety. The general rule does nothing except recurse, but it has a dependency which may be defined in the specific makefiles as the target of a rule to delete additional files:
  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: (none)
  - `ftn`: The debug and optimized static FORTRAN object libraries for the package, and also certain `BUILD.*` lock files which are normally deleted after a library is built but may be left behind if the build is terminated with prejudice.
  - `imp`: The debug and optimized static C++ object libraries for the package, dependency lists and the timestamp files maintained for the module by the dependency analysis (their deletion causes the dependency analysis to be done again from scratch), the `version.o` object module, the debug and optimized C++ template repositories (if any, in their entirety).
  - `ins`: (none)
  - `pkg`: Recursively delete the `tmp` directories for the package. These exist as subdirectories of `$(ARCHTMPD)`.
  - `scr`: (none)
  - `top`: (none)
  - `tst`: Test executables and reports in `$(BINTESTD)` generated by the `runtests` rule (see section §12.2, p252). Also, the intermediate object library produced when compiling all test executables in the directory.
- `cleandocs` : (specific)
  Clean up `$(AIPSDOCS)`.
  - `imp`: Delete defunct and unrecognized files from the corresponding subdirectory of `$(AIPSDOCS)`.
- `dependsys` : (specific)
  Update the system dependency lists in `$(ARCHAUXD)`. Since these lists are included by the makefiles they are normally updated automatically by *gmake*.
  - `imp`: Build the system dependency lists for the C++ libraries.
- `instsys` : (specific)
  Generate and possibly compile template instantiation files specified by a `templates` file in `$(CODEDIR)`.
  - `app`: Generate template instantiation files using *mkinst* (p180) and deposit them in subdirectory `$(CODEDIR)/tmplinst`.
  - `imp`: Generate template instantiation files using *mkinst* and deposit them in subdirectory `$(CODEDIR)/tmplinst`, then invoke `gmake allsys` in that directory to compile them and insert them into the object library.

- tst: Generate template instantiation files using *mkinst* and deposit them in subdirectory `$(CODEDIR)/tmplinst`.

- `version` : (specific)
  Compile `version.o`.
  - `imp`: Generate `version.o` using *doover* (p124), compile it and insert it into the object libraries.

- `lib(%.o)` : (specific, pattern rule)
  Compile an object module from the corresponding source file in `$(CODEDIR)` and insert it into the system library, then apply `$(RANLIB)` to the library.

  An AIPS++ installation may be configured to maintain a debug version of the object libraries, an optimized version, or both as determined by the setting of the *library control* variables in `makedefs` (p155).

  If dual debug/optimized libraries are being maintained then both forms are compiled and the libraries updated in tandem to ensure that they remain synchronized (see *updatelib* (p183)).

  Intermediate object files are stored in a subdirectory of `$(ARCHTMPD)`.
  - `ftn`: Compile the `.f` file for a FORTRAN subroutine and insert it into the FORTRAN library.
  - `imp`: Compile the `.cc` file for a C++ class implementation and insert it into the C++ library.

- `lib` : (specific)
  Invoke the `lib(%.o)` pattern rule for all sources in `$(CODEDIR)` to build the system object libraries (debug and/or optimized).
  - `ftn`: Build the FORTRAN library(s).
  - `imp`: Update the package library for all C++ sources in this module. Also, install any Glish scripts in `$(LIBEXECD)`.
  - `pkg`: Invoke `allsys` in the `implement` and `fortran` subdirectories.
  - `tst`: Invoke `allsys` in the `implement` directory for this package.

- `updatelib` : (specific)
  Insert all object modules into the system libraries and *ranlib* them.
  - `ftn`: Update the FORTRAN libraries.
  - `imp`: Update the C++ libraries.

- `ranlib` : (specific)
  Apply `$(RANLIB)` to the system static object libraries.
  - `ftn`: *ranlib* the FORTRAN libraries.
  - `imp`: *ranlib* the C++ libraries.

- `bin/$(AIPSEXES)` : (specific)
  Compile an executable in optimized mode from the corresponding source file in `$(CODEDIR)` and install it in `$BINOPTD`.

  - `aps`: The apps directory makefile generates a list of all applications which may be compiled from applications subdirectories. It then forms this into a list of targets of a rule for invoking the `bin` target in each applications subdirectory.
  - `ins`: Compile and install a utility program, e.g. *getrc* (p43).
  - `tst`: Synonym for `bintest/%`.

- `bindbg/$(AIPSEXES)` : (specific)
  Compile an executable in debug mode from the corresponding source file in `$(CODEDIR)` and install it in `$(BINDBGD)`.

  - `aps`: The apps directory makefile generates a list of all applications which may be compiled from applications subdirectories. It then forms this into a list of targets of a rule for invoking the `bindbg` target in each applications subdirectory.

- `bintest/%` : (specific, pattern rule)
  Compile a test program.

  - `tst`:
    Compile a test program from the corresponding source file in `$(CODEDIR)` and install it in `$BINTESTD`. If necessary, template instantiation files will be created from a `templates` file and compiled into an object library (which is retained). The compilation will be done in optimized mode if the `TESTOPT` `makedefs` variable is set to "`opt`", anything else will produce a debug compilation. The include path will be augmented with the name of the test subdirectory thereby allowing test programs to maintain a set of include files specific to them.

- `bin` : (specific)
  Compile all executables with sources in `$(CODEDIR)` in optimized mode and install them in the `$(BINOPTD)`.

  - `app`: Compile and install the application. The include path will be augmented with the name of the applications subdirectory thereby allowing an application to maintain a set of include files specific to it. Also, install any Glish scripts in `$(LIBEXECD)`.
  - `aps`: Compile and install all applications by invoking the `bin` target in every applications subdirectory.
  - `ins`: Compile and install all utility programs.
  - `pkg`: Invoke the `bin` target in the `apps` subdirectory.
  - `scr`: Synonym for `allsys`.
  - `tst`: Synonym for `bintest`.

- `bindbg` : (specific)
  Compile all executables with sources in `$(CODEDIR)` in debug mode and install them in the `$(BINDBGD)`.

  - `app`: Compile and install the application in debug mode. The include path will be augmented with the name of the applications subdirectory thereby allowing an application to maintain a set of include files specific to it.
  - `aps`: Compile and install all applications in debug mode by invoking the `bindbg` target in every applications subdirectory.

- `bintest` : (specific)
  Compile and install test programs.
  - `tst`: Invoke the `bintest/%` pattern rule for each test program in `$(CODEDIR)`. Delete the intermediate object library which may be produced for template instantiations created from a `templates` file.
- `docsys` : (general/specific)
  Compile and install documentation. Generally this invokes the `docscan` and `docextr` targets in the current directory as controlled by the `DOCEXTR` variable in `makedefs`. Specific makefiles may then do additional document compilations:
  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: Compile all `*.texi`, `*.latex` and `*.tex` files in `$(CODEDIR)` into `*.ps` PostScript files and install them in the corresponding `$AIPSDOCS` subdirectory.

    Files included by the `*.texi`, `*.latex` or `*.tex` files will be obtained from the corresponding `.dir` subdirectory. Any LaTeX-related `.bib` bibliography files contained in this directory will be processed by BibTeX.

    Plain-text `.text` files are simply copied to the `$AIPSDOCS` subdirectory, as are `.txt`, `.html`, `.htm`, `.gif`, and `.jpg` files.

    Any `<!`*Date*`>` meta-markups in `.html` and `.htm` files are parsed to record the time of installation.
  - `ftn`: (none)
  - `imp`: (none)
  - `ins`: (none)
  - `pkg`: (none)
  - `scr`: (none)
  - `top`: Recurse into the `doc` subdirectory.
  - `tst`: (none)
- `docscan` : (general/specific)
  Scan C++ sources building the database used by the document extractor. The general rule does nothing. Specific makefiles behave as follows:
  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: (none)
  - `ftn`: (none)
  - `imp`: Run *cxx2html* in *-scan* mode for each header file then recurse into all but `test` subdirectories.
  - `ins`: (none)
  - `pkg`: Recurse into the `implement` subdirectory.

  - `scr`: (none)
  - `top`: Recurse into all package subdirectories.
  - `tst`: (none)

- `docextr` : (general/specific)
  Extract documentation from C++ sources. The general rule does nothing. Specific makefiles behave as follows:
  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: (none)
  - `ftn`: (none)
  - `imp`: Run ***cxx2html*** for each header file to extract documentation from the C++ sources and install it in the corresponding subdirectory of `$AIPSDOCS`. Copy any `.gif` files to the same place. Then recurse into all but `test` subdirectories.
  - `ins`: (none)
  - `pkg`: Recurse into the `implement` subdirectory.
  - `scr`: (none)
  - `top`: Recurse into all package subdirectories.
  - `tst`: (none)

Notes

- The implement makefile includes the system dependency lists and so may start slowly as it checks to see whether the list is up-to-date (regardless of whether the particular target uses it or not). Inclusion of the dependency list can be circumvented by setting the `NODEP` variable (to anything). This causes the makefile to start considerably faster.

  `NODEP` is set automatically if ***gmake*** (p154) is invoked from a directory which does not reside under `$AIPSROOT`, that is, a programmer invokation. However, system-oriented targets may also be invoked from a programmer directory, and this would cause the dependency analysis to be circumvented, and the dependency list to be ignored. This may be legitimate if the target invoked does not actually use the dependency list, for example `chkout`, `cleancode` or `cleansys`. In fact, the code distribution system sets `NODEP` explicitly when it invokes the top-level makefile for these recursive targets for the system (see ***inhale*** (p78)). However, `NODEP` must not be set for an invokation of the implement makefile for a target which does use the dependency list. A proper resolution of this problem would require the facility for a makefile to examine its target list, but this is not currently possible.

- If the AIPS++ RCS source code repositories are present in an installation all AIPS++ makefiles access them via a symbolic link `$AIPSROOT/rcs` which is usually set to point to `slave`. Therefore, the slave RCS repositories are normally the ones consulted. However, it is possible to reset the `rcs` "switch" to `master` to cause ***gmake*** (p154) to checkout and/or rebuild AIPS++ from the `master` repositories. ***exhale*** (p73) uses this mechanism when constructing a new base release. However, it requires good bandwidth to the master and is only feasible in Socorro.

Examples

The following command suffices to clean up the AIPS++ system and rebuild it in its entirety

```
cd $AIPSCODE
gmake cleansys allsys
```

To just compile and install the documentation:

```
cd $AIPSCODE
gmake docsys
```

This would recursively compile all PostScript documents from the `*.latex`, `*.tex` and `*.texi` files in the `$AIPSCODE/doc` directory tree and install them in the corresponding `$AIPSDOCS` subdirectory. ASCII, `*.html`, `*.ps` etc. documents will also be copied across. The document extractor would then be invoked on the C++ sources (if enabled). Note, however, that no documents are printed via `docsys`.

See also

The GNU `Make` manual.
The GNU manual page for *gmake*.
The unix manual page for *ranlib*(1).
AIPS++ variable names (§1.2, p8).
gmake (p154), GNU make.
`makedefs` (p155), AIPS++ makefile definitions.
updatelib (p183), update an AIPS++ object library.

10.10 *mkinst*

Generate template instantiation files from a list.

Synopsis

mkinst [-*q*] package

Description

mkinst generates a template instantiation file for each entry in the specified file, (`templates` by default). It leaves the resulting `.cc` files in the current directory. The format of the entries in the `templates` file is

```
<ident> <.cc file> <.h file> [<extra .h files>] <template declaration> ...
```

or

```
<ident> <.h file> [<extra .h files>] <template declaration> ...
```

The latter form is for instantiation of purely inline functions. The `.cc` and `.h` files are expressed as relative pathnames with respect to `$AIPSROOT/code/include`. For example:

```
1050 aips/Set/Set.cc template class Set<Int>
1040 aips/Set/Set.cc aips/Cube/Cube.h template class Set<Cube>
1060 aips/Set/Set.cc template Set<Int> operator+(Set<Int> const &, Set<Int> const &)
1060 aips/Set/Set.cc template void union(Set<Int> const &, Set<Int> const &)
```

The `ident` contains a 4-digit number which is unique amongst all templates based on a particular class implementation `.cc` or `.h` file. The name of the template instantiation `.cc` file is composed of the name of the class `.cc` or `.h` file and the `ident`, for example, the instantiation file for the first entry above would be `Set_1050.cc`.

Comments in the `templates` file are signalled by a "#" in the first field. Blank lines are also allowed.

Each entry in the `templates` can be split over several lines. The first line must have the `ident` and one or more of the necessary include files (.cc or .h). Subsequent lines can be in any order, and have one of the following formats:

```
= file.h  additional include files
template class Name<...> class specialisation
template <...> ...  templated class forward declaration
template <> ...  non-templated forward declaration
template ret-val function(....)  function specialisation
typedef definition name              typedef, for use in next
AIPS_[A-Z0-9]*_AUX_TEMPLATES(x, ...) to aid with sub-templates
```

The *AIPS_()* macros are provided to aid in the generation of auxilliary templates necessary for some compilers for standard library templates (like the map and vector e.g.). Since macros cannot handle arguments with embedded commas, the `typedef` option is provided to give macro arguments with commas. See the repositories for examples.

Template declarations can be bracketed by #if #else #endif. For example:

```
   1040 aips/Set/Set.cc aips/Cube/Cube.h
template class Set<Cube>
   1050 aips/Set/Set.cc
template class Set<Int>
template <> class ostream
template <class T> class Vector
#if !defined(STD_LIB)
       template Set<Int> operator+(Set<Int> const &, Set<Int> const &)
         template void union(Set<Int> const &, Set<Int> const &)
#endif
```

Any template instantiation `.cc` file in the current directory which does not correspond to an entry in the templates file will be deleted. These files are recognized by having names of the form `*_[0-9][0-9][0-9][0-9].cc`.

Options

-q Run quietly.

Notes

- The `templates` file is normally sorted on the name of the `.cc` or `.h` file so that it can easily be checked that the idents are unique. A separate utility, ***reident*** (p214), is provided to help manage the idents.
- If a `templates` file entry is changed then its `ident` should also be changed to force its instantiation file to be regenerated.

Diagnostics

Status return values
  `0`: success
  `1`: the `templates` file contains one or more bad entries

Examples

Invokation of ***mkinst*** is usually done indirectly via specific makefile targets (see §10.9, p171 and §11.4, p199). These maintain the template instantiation files in a subdirectory (`tmplinst`) of the directory where the `templates` file resides.

```
mkdir tmplinst
cd tmplinst
mkinst -q ../templates
```

See also

reident (p214), Remake the idents in a `templates` list.
duplicates (p216), Checks for duplicate entries in `templates` lists across the system.

Author

Original: 1996/04/17 by Mark Calabretta, ATNF after a *perl* script by Shelby Yang, NRAO.

10.11 *updatelib*

Update AIPS++ object libraries.

Synopsis

updatelib [-*f*] package

Description

updatelib updates the object libraries associated with a particular AIPS++ package and language (see §1, p1). It is normally used only by AIPS++ `makefiles` (p162) but can be invoked in stand-alone mode. It can process dual debug and optimized libraries concurrently with another invokation, possibly on another machine.

An "`_f`" or "`_c`" appended to the package name, for example "`aips_f`", indicates that the FORTRAN or C libraries for the package are to be updated rather than the default C++ libraries. The libraries themselves have names of this form, for example `$LIBOPTD/libaips_f.a`.

The `lib` and `lib/%` targets of the implement and fortran makefiles deposit object modules in package/language-specific subdirectories of `$ARCHTMPD` referred to as `$TMPPCKGD`, for example `$ARCHTMPD/aips_f`. These have separate `dbg` and `opt` subdirectories for debug and optimized compilations. When the `lib` rules have finished compiling all modules they invoke ***updatelib*** to insert them into the libraries.

Once invoked, ***updatelib*** copies the library into the temporary area, inserts the object modules into it, and then ***ranlib***s it. It does this in turn for the debug and and optimized libraries.

updatelib uses the library control variables to determine whether sharable libraries are to be built (see `makedefs` (p155)). If so, they are constructed in a subdirectory called `BLDSHR.$ID`, where `$ID` is an identification string of the form `<user>@<host>`.

Only if each static and sharable library was successfully updated does it copy them back to `$LIBDBGD` and/or `$LIBOPTD`.

With reference to the library control variables, ***updatelib*** explicitly looks for and deletes static and/or shared libraries for the package/language which are not being maintained, thus ensuring that the libraries remain synchronized.

This whole process is complicated by the fact that the debug and optimized libraries for a package-language combination can be updated concurrently by multiple invokations of ***updatelib***, possibly on different machines. Independent invokations of ***updatelib*** communicate via semaphore files with names of the form `updatelib.*`. These are described below.

Options

-f Force a ***ranlib*** and rebuild of the sharable objects (if applicable) even if no new object modules are found.

Notes

- When building the system the implement makefile recurses through all module subdirectories, compiling object modules and accumulating them in a temporary directory. The object library is only updated after all modules have been compiled. The alternative of updating the object library after processing each software module imposes a large overhead because of the large size of the object library, particularly the debug version.

  However, the large number of object modules in some AIPS++ packages may cause problems for ***updatelib*** simply by causing the argument list to exceed the maximum imposed by the unix kernel. This is particularly troublesome for systems which use do-it-yourself template instantiation (§10.8, p168), or have a small value of `ARG_MAX` in `limits.h` (e.g. HPUX).

  To circumvent this ***updatelib*** recognizes an `UPDBATCH` makedefs variable which causes it to process object modules in batches no bigger than the specified size.

- ***updatelib*** should not be terminated with a `SIGKILL` ("`kill -9`") since this would prevent it from cleaning up its semaphore files and possibly cause library updates to be locked out for 24 hours. ***updatelib*** exits gracefully on receiving a `SIGHUP` ("`kill -1`") or a `SIGTERM` ("`kill -15`").

- Stand-alone invokations of ***updatelib*** obtain `makedefs` variable definitions via the `eval_vars` target.

Diagnostics

Status return values
`0`: success
`1`: initialization error
`2`: library update failed

If any library update fails, none of the libraries will be updated—with one exception: if a static library build succeeds and its corresponding shared library build fails then the static library will be updated.

Examples

The command

```
updatelib aips
```

would update `$LIBDBGD/libaips.a`, `$LIBOPTD/libaips.a`, and the corresponding sharable libraries if enabled for the installation.

Files

updatelib uses the following semaphore files:

`$TMPPCKGD/updatelib.process.$ID` ...This file simply declares an invokation of ***updatelib***. ***updatelib*** determines whether it is the first invokation by testing for the existence of any file with a name of the form `updatelib.process.*`.

`$TMPPCKGD/updatelib.proceed` ...This file is created by the first invokation of ***updatelib***. It is used as an "all-is-well" semaphore and is deleted by any invokation which fails to update a library. Other invokations abort their updates if it disappears.

`$TMPPCKGD/{dbg,opt}/updatelib.$ID` ...This is a lock file for the task of updating the static library, whether debug or optimized, and possibly its sharable counterpart. Files older than 24 hours are ignored when testing for the existence of one of these locks. If a valid lock does already exist then ***updatelib*** reports it before looking for further work.

`$TMPPCKGD/{dbg,opt}/updatelib.done` ...When an invokation of ***updatelib*** has finished updating the library it renames `updatelib.$ID` to `updatelib.done` and then looks for further work. The invokation which has reserved the job of installing the updated libraries tests for the existence of this file as a signal that the invokation responsible for updating the library has finished.

`$TMPPCKGD/updatelib.lock` ...Lock file for the invokation of ***updatelib*** which will install the updated libraries. If an invokation sees that this file already exists it waits for it to disappear before proceeding, thereby synchronizing the termination of multiple invokations and ensuring that ***updatelib*** doesn't return before the library update is complete.

The invokation of ***updatelib*** which installs the libraries also deletes all semaphore files on completion.

See also

The unix manual page for ***ar***(1).
The unix manual page for ***ranlib***(1).
AIPS++ variable names (§1.2, p8).
`makedefs` (p155), AIPS++ makefile definitions.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

Author

Original: 1992/04/03 by Mark Calabretta, ATNF.

Chapter 11

Code development

Information specifically for programmers who are developing code in AIPS++ [1].

11.1 Overview of code development

This section runs through the basic operational aspects of developing code in AIPS++.

Setup

Basic AIPS++ programmer setup was described in §3.1.2, p20. To recapitulate the main points

- If you will be checking sources into and out of the AIPS++ master (i.e. an AIPS++ consortium programmer) then make sure you are a member of the AIPS++ programmer group, the conventional name of which is *aips2prg*.
- Modify your shell's login script (`.login`, `.profile`, etc.) to invoke ***aipsinit*** (p31) to define the AIPS++ environment.
- Invoke ***mktree*** (p126) to create an AIPS++ workspace.

The AIPS++ makefiles normally deposit binary files in the same directory as programmer source files unless you create separate directories for them. If you intend to compile sources for more than one architecture or compiler then you should create these directories. A good strategy is to maintain sources within your home directory (which presumably is backed up) and relegate the binaries to some other filesystem with more space but not backed up. Hence typically

```
mkdir /bigdisk/myarea/sun4sol_gnu/{aux,bin,bindbg,lib,libdbg}
ln -s /bigdisk/myarea/sun4sol_gnu $HOME/aips++/sun4sol_gnu
```

Do this for each AIPS++ architecture that you are likely to use. The AIPS++ makefiles will discover and use these directories automatically. You can use ***aipsinit*** (p31) to switch between architectures at any time.

[1] Last change: Id: CodeDevl.tex,v 19.4 2004/10/18 02:31:45 wbrouw Exp

Modifying existing sources

Once your workspace is set up the AIPS++ makefiles allow you to compile system sources without having to check them out into your workspace. Programmer-oriented makefile targets are described in §11.4, p199. Typically it is enough to invoke *gmake* (p154) without any targets. Use

```
gmake help
```

to get a context-sensitive summary of the AIPS++ targets for each directory. Programmer-oriented targets are discussed in section §11.4, p199.

If you wish to modify a source file then

- If you are a consortium programmer and intend to change the master source code then check out the relevant source file using *ao -l* (p97) or *rao -l* (p133). If you do not intend to update the master then omit the *-l* flag.
- If you are not a consortium programmer or only wish to work within your local system then
  - If a copy of the RCS repository is maintained in your installation (as indicated by the presence of the `$AIPSROOT/rcs` directory) and you intend to update your installation then check out the source file using *co -l*. If you do not intend to update the local system then omit the *-l* flag.
  - Otherwise, if your installation does not contain a copy of the RCS source repository then simply copy the relevant system file to your workspace directory.

Once you've modified the file you can compile it in your workspace without affecting the system binaries. When you're satisfied with your changes

- If you want to update the AIPS++ master check the file in with *ai* (p93) or *rai* (p130). If you also want to update the local slave RCS repository then do so with *au* (p109) or *rau* (p136). You can combine the two operations via *ai -au* or *rai -au*.
  Changes need to be documented in the AIPS++ changelog file using *ac* (p89) or *rac* (p129).
- Use *ci* if you only want to update the local RCS repository.
- If your installation does not contain a copy of the RCS source repository then simply copy the file to the system directory.

If you want to update the system binaries you will need to issue a

```
gmake allsys
```

This can be done from a programmer workspace (although some installations explicitly prevent this by removing write permission for *aips2prg* from the AIPS++ system directories).

The gmake command line - flags and variable definitions

gmake (p154) has several command line options which are frequently useful for AIPS++ programmers.

- *--help*: Print usage information and exit without processing any makefiles.
- *-i*: Ignore all errors in commands executed to remake files.
- *-k*: Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same.
- *-n*: Print the commands that would be executed, but do not execute them.
- *-W*: Pretend that the target file has just been modified. When used with the *-n* flag, this shows you what would happen if you were to modify that file. Without *-n*, it is almost the same as running a *touch* command on the given file before running *gmake* (p154), except that the modification time is changed only in the imagination of *gmake*.

Note that the default `makedefs` (p155) sets the *--no-print-directory* internally (via `MAKEFLAGS`) and also sets *-i* for system invokation but not for programmer invokation. Consult the unix manual page for *gmake* for further options or use

```
gmake --help
```

Do not confuse this with

```
gmake help
```

which is an AIPS++ target which lists AIPS++ targets. Note also that you should eschew usage of the *-e* option (environment variables override makefile variables) unless you fully understand its implications.

The other important aspect of *gmake* invokation is that makefile variables may be defined on the command line and these (usually) override their definition within the makefile. The makefiles use separate sets of variables for system and programmer compilations, so for example, system C++ compilations use `C++DBG` and `C++OPT` for debug and optimized compilations whereas `C++FLAGS` is used for programmer compilations. The default `makedefs` (p155) sets

```
ifdef OPT
   C++FLAGS  = $(C++OPT) $(EXTRA_C++FLAGS)
else
   C++FLAGS  = $(C++DBG) $(EXTRA_C++FLAGS)
endif
```

although as described in section §11.3, p196 the site-specific makedefs may redefine this, for example as

```
ifdef OPT
   C++FLAGS := -O$(OPT) -pipe
else
   C++FLAGS := $(C++DBG)
endif

C++FLAGS += $(EXTRA_C++FLAGS)
```

Either way, this example shows three basic types of command-line variable substitutions

- Variables such as `OPT` which are used as switches. In the second example it is used as a switch and an optimization level. It might be used as

  ```
  gmake OPT=2 myprog
  ```

- The `C++FLAGS` variable itself can be completely redefined on the command line, for example

  ```
  gmake C++FLAGS="-O4 -fast -target=native" myprog
  ```

- The `EXTRA_C++FLAGS` variable can be used to add extra options to `C++FLAGS`, for example

  ```
  gmake EXTRA_C++FLAGS="-fast" myprog
  ```

As discussed in section §12.2, p252, a list of variables of specific interest to programmers may be obtained via

```
gmake show_prg
```

This may be used to show the value of the variables which would be used in a particular situation, for example

```
gmake OPT=2 EXTRA_C++FLAGS="-fast" show_prg
```

The variables of interest to programmers are described in more detail in section §11.4, p199.

As suggested above, the site-specific `makedefs` (p155) may be used to alter the construction of programmer variable definitions. Several conventional constructions are discussed in section §11.3, p196. Programmers may also implement their own definitions or constructions in a personal `makedefs` file as discussed in the same section.

Creating new files and directories

If you want to develop a new directory in the AIPS++ hierarchy, usually to contain a class implementation module or a new application, then simply create that directory in your workspace. Likewise, new files may simply be created - the AIPS++ makefiles do not contain lists of files but use the file suffix to determine how to process a file. Recognized file suffixes are listed in section §10.6, p162.

All AIPS++ directories (except for `.dir` and `tmplinst` directories) must contain a makefile which must be called `makefile`. Typically this need only include one of the generic AIPS++ makefiles (see §10.6, p162). For example, when creating a new application the minimum requirement would be

```
# $Id$

# Use the generic AIPS++ application makefile.
#---------------------------------------------
include $(word 1, $(AIPSPATH))/code/install/makefile.app
```

Once you have finished development the new directory may be checked in to the master via ***ai*** (p93). However, you may receive a message to the effect that permission for the operation is denied - creation of new files and directories in many parts of the master is restricted to particular individuals since all new code must be vetted by the "Code-Cop".

makefiles

The makefiles which exist in the AIPS++ source directories mostly just include a generic makefile but some need to supply additional information.

For an application you often need to register any required external libraries and may also want to define some preprocessor directives as described in section §10.7, p165:

```
# $Id$

# Use the generic AIPS++ application makefile.
#---------------------------------------------
XLIBLIST := GLISH WCSLIB F77
CPPMODU  := -DFOO -DBAR
include $(word 1, $(AIPSPATH))/code/install/makefile.app
```

The `XLIBLIST` and `CPPMODU` variables are relevant to the `app`, `imp`, and `tst` makefiles.

The documentation makefile requires arguments to be specified for conversion of LaTeX documents to HTML via ***latex2html***. Typically

```
# $Id$

# Declare latex2html flags.
System := -split 2
Coding := -split 3
```

```
# Use the generic AIPS++ documentation makefile
#-----------------------------------------------
include $(word 1, $(AIPSPATH))/code/install/makefile.doc
```

This example declares that *latex2html* should be run on `System.latex` and `Coding.latex`. No other files in the directory with a `.latex` suffix will be converted to HTML although they will be converted to PostScript. At the minimum the variable of the same basename as the `.latex` file should specify a `-split` argument to *latex2html*, but it can also specify any other arguments recognized by it.

This example also illustrates that LaTeX documents must have a `.latex` suffix to distinguish them from plain TeX documents.

Adjunct `makedefs` files may contain definitions (or, in principle, rules) which apply to a particular source code directory. The adjunct files are architecture-specific and are primarily intended to control compiler options on a per-sourcefile basis. This is an advanced topic discussed in section §10.5, p160.

See also

Section §1.4, p9, AIPS++ accounts and groups.

11.2 Coding conventions

This section discusses some aspects of the AIPS++ system which have a direct effect on the way source code is written.

Include files

It is essential in AIPS++ that the

```
#include <Header.h>
```

form of preprocessor `#include` statement be used. This form looks for `Header.h` in the search path defined by *-I* compiler options and allows the AIPS++ makefiles to control the include process. The other form admitted by the C preprocessor, namely

```
#include "Header.h"
```

should never be used in AIPS++ source code. It causes the preprocessor to look for the include file in the same directory as the source file, regardless of the search path defined by *-I* compiler options. This may cause problems where, for example, a programmer has checked out and is modifying a C++ class implementation file but doesn't have the relevant header files checked out alongside.

Include files are specified as in the following example:

```
#include <aips/Tables/Table.h>
```

This refers to the specific header file `$(AIPSCODE)/aips/implement/Tables/Table.h` by virtue of the symlink

```
$(AIPSCODE)/include/aips -> $(AIPSCODE)/aips/implement
```

and the fact that `$(AIPSCODE)/include` is added to the include path via a *-I* compiler option (see §1.1.1, p1). Symlinks for each package are created by the `symlinks` rule in the top-level makefile (§10.9, p171), and they are created in programmer workspaces by ***mktree*** (p126).

Another caveat on include files is that they *must* have a `.h` file suffix. This is mandated by the need for `vpath` directives in the makefiles, specifically for the dependency analysis used in programmer compilations.

perl scripts

perl scripts must begin with a line of the form

```
#!/usr/local/bin/perl4
```

or

```
#!/usr/local/bin/perl5
```

When the scripts are installed the makefile will replace these strings with the full pathname to the *perl4* or *perl5* executable encoded in the PERL4 and PERL5 makedefs variables. *perl* options may be added after these strings in the usual way.

Documentation include files

Files included by TeX, LaTeX and Texinfo source files may be deposited in a subdirectory with the same basename as the source file but with a .dir suffix. For example, the source for this manual consists of System.latex which simply defines the document style and a few commands and environments and then includes a separate .tex file for each chapter. System.latex resides in $AIPSCODE/doc/reference and the chapters reside in $AIPSCODE/doc/reference/System.dir.

The documentation makefile automatically adds any .dir subdirectory to the TEXINPUTS environment variable which defines TeX's search path for included files. Note that, while .dir subdirectories can contain any type of file, TeX, LaTeX and Texinfo may themselves impose constraints, for example LaTeX requires a .tex file suffix for included files.

The .dir subdirectories should not contain a makefile, nor should they have subdirectories. Other than that there is really no restriction on what they may contain.

URLs in html documents

Because AIPS++ HTML documents are propagated to many sites and installed with minimal changes it is important that the URLs (Uniform Resource Locators) in these documents be relative to $AIPSDOCS whenever possible. Someone navigating the AIPS++ documentation at one site should not be directed to another AIPS++ installation without needing to be. Once such a traversal is made the error is compounded since the links stay remote unless the link stack is unwound.

A document in a subdirectory of $AIPSCODE/doc is installed within the corresponding subdirectory of $AIPSDOCS. For example, $AIPSCODE/doc/html/homepage.html is installed in $AIPSDOCS/html/homepage.html. Thus a link from homepage.html to, say, $$AIPSDOCS/projects/mission.html should be given as

```
<A HREF="../project/mission.html">Mission Statement</A>
```

The following prescription should be observed:

- *Use a relative URL* if the linked-to file exists (or ought to exist) in all AIPS++ installations.
- *Use an absolute URL* if the linked-to file does not exist in all AIPS++ installations. Typically these would be links to web services provided by the master as described in §13.1, p263.

HTML meta-markups

The HTML comment

```
<!$Date$>
```

may be used to automatically record timestamp information in the installed HTML file. The following fragment shows an example of its use, typically near the end of the file:

```
<P>Copyright &#169; 1996-98,2000,2001,2004 Associated Universities Inc., Washington, D.C.</P>
<ADDRESS>aips2adm@nrao.edu</ADDRESS>

<!$Date$>

</BODY>
</HTML>
```

RCS substitutes the current time (GMT) for the `$Date$` string when the file is checked in (see ***ai*** (p93)) thereby transforming the meta-markup to something resembling

```
<!$Date: 1995/08/04 05:22:48 $>
```

The documentation makefile parses this into HTML and also adds the time when the `.html` file was installed in the relevant subdirectory of `$AIPSDOCS` (see §1.2, p8). Within an HTML browser the result is as follows:

```
Copyright (c) 1995-98,2000,2001,2004 Associated Universities Inc., Washington, D.C.

aips2adm@nrao.edu

---------------------------------------------------------------------------

Modified:  1995/08/04 05:22:48 GMT
Installed: 1995/08/06 19:32:08 GMT
```

It is permissable to insert additional text within the meta-markup as follows:

```
<!$Date$ by mcalabre@atnf.csiro.au>
```

This would appear as:

```
Copyright (c) 1995-98,2000,2001,2004 Associated Universities Inc., Washington, D.C.

aips2adm@nrao.edu

---------------------------------------------------------------------------

Modified:  1995/08/04 05:22:48 GMT by mcalabre@atnf.csiro.au
Installed: 1995/08/06 19:32:08 GMT
```

The `<!$Date$>` meta-markup must appear as the first and only text on a line (with no leading whitespace). It should be the last item before the `</BODY>` HTML markup.

11.3 Code development makedefs conventions

Conventions for `makedefs` variable definitions

Some features of the AIPS++ programmer compilations are defined as conventions for variable usage which are implemented within `makedefs` (p155) itself.

Optimized programmer compilations:

By default, programmer compilations are usually done in debug mode. Specifying `OPT=1` (or `2`, `3`, `4`, ...) on the *gmake* (p154) command line signals that programmer compilations are to be done at the corresponding optimization level. For example

```
gmake OPT=1 Foo.o
```

If the `OPT` variable is defined the default `makedefs` (p155) will set all programmer compile options to be the same as used for optimized system compilations. However, alternate levels of optimization must be implemented in the site-specific `makedefs`; the example below shows how this might be done. If `OPT` has been set then programmer executables will be linked to the optimized system libraries (if they exist).

The `OPT` variable is not intended to (and should not be made to) affect system compilations.

Alternate programmer compilation options:

Another convention in use for programmer compilations is that of setting compiler `*FLAGS` variables on the command line to `alt1`, (`alt2`, ...). For example,

```
gmake C++FLAGS=alt1 Foo.o
```

This is a signal to the site-specific `makedefs` (p155) to substitute a predefined, locally standard, set of compiler options. By convention, `alt1` should be defined to do an optimized compilation of the same level as the system optimized compilation.

Adding extra programmer compilation options:

Programmers may supplement the compilation options defined within `makedefs` (p155) by use of `EXTRA_*FLAGS`, for example

```
gmake EXTRA_C++FLAGS="+p -pipe" Foo.o
```

which would cause "`+p -pipe`" to be appended to the value of `C++FLAGS` defined within `makedefs`. If any of the extra options are incompatible with, or contradict the options set in `makedefs` then compiler-specific behaviour will result.

Redefining programmer compilation options in full:

Strictly speaking this is not a `makedefs` (p155) convention, but nevertheless it's worth pointing out that `makedefs` must not prevent programmer variables from being redefined in full on the *gmake* (p154) command line. In practice this simply means that `makedefs` must not use the `override` directive when defining programmer compilation variables (except in implementing the alternate options as described below).

Implementing the conventions:

The conventions on programmer compilation options must be implemented within the site-specific `makedefs` (p155). However, they need not be implemented if they are not required. Since considerable latitude is available in their precise interpretation a `show_prg` target has been provided for programmers to print the values of all variables likely to be affected. The following fragment shows a realistic example of how the `C++FLAGS` variable might be defined in the site-specific `makedefs`:

```
ifdef OPT
   C++FLAGS := -ptr$(PGMRPTRD) -pta -O$(OPT) +p -pipe -ptv
else
   C++FLAGS := -ptr$(PGMRPTRD) $(C++DBG)
endif

ifeq "$(C++FLAGS)" "alt1"
   override C++FLAGS := -ptr$(PGMRPTRD) $(C++OPT)
endif

ifeq "$(C++FLAGS)" "alt2"
   override C++FLAGS := -ptr$(PGMRPTRD) $(C++DBG) -ptv
endif

C++FLAGS += $(EXTRA_C++FLAGS)
```

An important point to note here is that if `C++FLAGS` is defined on the command line then a redefinition within `makedefs` (p155) has no effect unless the `override` directive is used. The variables which define programmer compilation options are `CPPFLAGS`, `CFLAGS`, `C++FLAGS`, `FFLAGS`, and `LDFLAGS`.

Programmer binary directories:

The `PGMRINCD`, `PGMRAUXD`, `PGMRLIBD`, and `PGMRBIND` variables specify where the output from programmer compilations is to be deposited. `PGMRINCD` is used for the output of the ***lex*** and ***bison*** parser/generators, and `PGMRAUXD` contains the results of programmer dependency analyses. The default `makedefs` (p155) sets these, and also `PGMRPTRD`, to

```
PGMRINCD := $(wildcard $(PGMRARCH))
PGMRAUXD := $(wildcard $(PGMRARCH)/aux)
PGMRLIBD := $(wildcard $(PGMRARCH)/lib)
PGMRBIND := $(wildcard $(PGMRARCH)/bin)
PGMRPTRD  = $(PGMRARCH)/ptrepository
```

In other words, the `$(PGMRARCH)`, `$(PGMRARCH)/aux`, `$(PGMRARCH)/lib`, and `$(PGMRARCH)/bin` directories will be used *if they exist.* If they don't, then `PGMRINCD`, `PGMRLIBD` and/or `PGMRBIND` will be blank and the corresponding output will be left in the programmer's source code directories. The site-specific `makedefs` (p155) files can reset these variables as it sees fit, except that if they're not blank, then they must specify absolute pathnames. Note that this facility uses the ***-o*** compiler option for object modules so `PGMRLIBD` must not be set for compilers which do not support it.

The site-specific `makedefs` may wish to redefine `PGMRLIBD` and `PGMRBIND` depending on the value of `OPT` in order to separate the binaries produced for different levels of optimization. The default is to put debug and optimized binaries in the same place.

Programmer makedefs

Programmers can maintain makefile variable definitions (and rules) within a private `makedefs` file. This is included after the site-specific `makedefs` so that it's possible to override any site definitions. The programmer `makedefs` must reside in the architecture-specific subdirectory of the root of the programmer's workspace. For example, if the AIPS++ architecture is `sun4sol_gnu` and the programmer's workspace is rooted at `$HOME/aips++` then the programmer `makedefs` must be `$HOME/aips++/sun4sol_gnu/makedefs`. Of course, this file could include `makedefs` definitions from elsewhere.

See also

The GNU `Make` manual.
The GNU manual page for *gmake*.
AIPS++ variable names (§1.2, p8).
gmake (p154), GNU make.
`makedefs` (p155), AIPS++ makefile definitions.

11.4 Code development makefile rules

Use of the AIPS++ makefiles for code development.

Synopsis

```
makefile
makefile.{app,aps,chk,doc,ftn,imp,pkg,scr,tst}
```

Description

The AIPS++ code development targets are listed below by category. These lists are not exhaustive, but do aim to cover everything of practical use. In particular, they omit targets which are intended for the internal use of the makefiles.

A target is labelled as "recursive" if it causes ***gmake*** (p154) to be invoked in all subdirectories. It is "general" if it applies to all makefiles; such targets are defined in `makedefs` (p155). A target is "specific" if defined in a specific makefile.

Some targets such as `allsys` have a general meaning, the specific behaviour of which differs for specific makefiles. These are referred to as "general/specific" and where appropriate the details of a target's behaviour are described for each of the generic makefiles, for the top-level makefile (`top`), and the installation makefile (`ins`).

Targets which apply only if the RCS source code repositories are present are marked as "RCS".

Programmer-oriented variables:

Rules for programmer-oriented targets look for source files first in the programmer directory and if not found search the corresponding subdirectory of `$AIPSCODE` which is referred to below as `$(CODEDIR)`. This means that programmers can compile code in their own workspace without having to copy source files from `$(CODEDIR)`, thereby minimizing the number of files that need to be present in the programmer's workspace and reducing the possibility that these may be "stale".

Certain variables are defined in `makedefs` (p155) for the exclusive use of programmer-oriented rules and may be redefined as necessary on the ***gmake*** (p154) command line. These are:

- `PGMRINCD`
  Repository for include files produced by parser/generators such as ***lex*** and ***bison***.

  The default `makedefs` sets this to `$PGMRARCH` if that directory exists (but the site-specific `makedefs` may redefine it).

  Output is left in the current directory if `$PGMRINCD` is blank.

- `PGMRAUXD`
  Repository for dependency lists and timestamp files generated by the programmer dependency analysis.

  The default `makedefs` sets this to `$PGMRARCH/aux` if that directory exists (but the site-specific `makedefs` may redefine it).

  Output is left in the current directory if `$PGMRAUXD` is blank.

- PGMRLIBD
  Programmer directory for class object modules and libraries.

  The default makedefs sets this to $PGMRARCH/lib if that directory exists (but the site-specific makedefs may redefine it).

  Output is left in the current directory if $PGMRLIBD is blank.

- PGMRBIND
  Directory where programmer executables are deposited; programmers should add this directory to their PATH.

  The default makedefs sets this to $PGMRARCH/bin if that directory exists (but the site-specific makedefs may redefine it).

  Output is left in the current directory if $PGMRBIND is blank.

- PGMRPTRD
  Directory serving as the programmer template repository.

  The default makedefs sets this to $PGMRARCH/ptrepository as a suggestion and for the convenience of the site-specific makedefs in setting compiler options. However, the site-specific makedefs may redefine it or simply ignore it, and PGMRPTRD is not used directly by any of the makefiles.

  Defaults to a compiler-specific directory if blank.

- EXTRA_PGMRINCL
  Extra include directories to be *prepended* to the internally defined include path, PGMRINCL, which may also be redefined.

- EXTRA_PGMRLIBS
  Extra object libraries to be *appended* to the internally defined library list, PGMRLIBS, which may also be redefined.

- OPT
  Optimization level to use for compilations. The exact meaning of this is defined in the site-specific makedefs file.

- OPTLIB
  This variable may be set (to anything) to force a debug programmer compilation to link to the optimized system libraries. This is useful for reducing the size of executables and the time taken to link them if you only wish to debug your own code.

- *FLAGS
  Options to be used for programmer compilations (CPPFLAGS, CFLAGS, C++FLAGS, FFLAGS, LDFLAGS and CXXARFLAGS).

- EXTRA_*FLAGS
  Extra compiler flags to be appended to *FLAGS (CPPFLAGS, CFLAGS, C++FLAGS, FFLAGS, LDFLAGS and CXXARFLAGS).

These variables may be printed with the show_prg target. The *FLAGS variables are the subject of a convention for setting alternate programmer compilation flags described in the section on 11.3.

Programmer-oriented targets:

The programmer-oriented targets are as follows; note that where targets are defined in terms of a *gmake* (p154) variable, for example `$(SUBDIRS)`, it, and all other variables, may be printed via the `show_all` target:

- `all` : (default programmer target, general/specific)
  This is declared generally in `makedefs` as the default target if *gmake* (p154) has *not* been invoked from a subdirectory of `$AIPSCODE`. Its dependencies and commands are defined in the specific makefiles to compile all object modules, executables, etc. from files in the current directory.
  - `app`: Compile this application and put the executable in `$(PGMRBIND)`.
  - `aps`: Compile all applications which have their main `.cc` file in the programmer's workspace and put the executables in `$(PGMRBIND)`.
  - `chk`: Does nothing.
  - `doc`: Compile all `info` files and PostScript documents with sources in the current directory.
  - `ftn`: Update all programmer object library modules which have sources in the current directory.
  - `imp`: Update all programmer object library modules which have sources in the current directory.
  - `ins`: Compile all utility programs with sources in the current directory.
  - `pkg`: Does nothing.
  - `src`: Does nothing.
  - `top`: Create the `include` subdirectory and populate it with symbolic links to the `implement` directories of all installed packages (see *mktree* (p126)).
  - `tst`: Compile all test programs with sources in the current directory and put the executables in `$(PGMRBIND)`.
- `$(SUBDIRS)` : (general)
  `makedefs` sets the `SUBDIRS` variable to the names of all subdirectories which have a makefile, thus making them recognized targets. If the RCS source code repositories exist then `$(SUBDIRS)` lists the subdirectories of the corresponding RCS directory. This list is then made the target of a rule for creating the subdirectory and its `RCS` symlink (see *mktree* (p126)), and initiating a *gmake* (p154) within it. Note that this target has a system-oriented counterpart which behaves similarly.
- `clean` : (general/specific)
  This target causes intermediate files to be deleted. It is defined generally in `makedefs` with a dependency which may be defined in the specific makefiles as the target of a rule to delete additional files. General files deleted are `*` , `*%`, `a.out`, and `core`. The specific makefiles may delete more:
  - `app`: Deletes `*` , `*%`, and `$(PCKGMOD).lock` from the `$(PGMRAUXD)` directory, and deletes `*.i`, `*.o`, `*.cdb`, `*.cyi`, `a.out`, and `core` from the `$(PGMRBIND)` directory.

- aps: Deletes * , *%, and $(PACKAGE)=*.lock from the $(PGMRAUXD) directory, and deletes *.i, *.o, *.cdb, *.cyi, a.out, and core from the $(PGMRBIND) directory.
- chk: (none)
- doc: *.aux, *.bbl, *.blg, *.cp, *.cps, *.dvi, *.fi, *.fis, *.fn, *.fns, *.idx, *.ilg, *.ind, *.ky, *.kys, *.lof, *.log, *.pg, *.pgs, *.toc, *.tp, *.tps, *.vr, *.vrs, *.info, *.info-*
- ftn: Deletes *.o a.out, and core from the $(PGMRLIBD) directory.
- imp: Deletes *.i, *.o, *.cdb, *.cyi, a.out, and core from the $(PGMRLIBD) directory, deletes * , *%, *.lcc, and *.ycc from the $(PGMRINCD) directory, and deletes * , *% and $(PCKGMOD).lock from the $(PGMRAUXD) directory.
- ins: *.i, *.o
- pkg: (none)
- src: (none)
- top: (none)
- tst: Recursively deletes *_tmp* from the current directory, deletes * and $(PCKGMOD).lock from the $(PGMRAUXD) directory, and deletes *.i, *.o, *.cdb, *.cyi, a.out, and core from the $(PGMRLIBD) and $(PGMRBIND) directories.

• cleaner : (general/specific)
This target is defined generally in makedefs with a dependency which may be defined in the specific makefiles as the target of a rule to delete additional files. Generally this target invokes the clean target and then deletes source files in the current directory which have been checked out without a lock. It determines these as any file in the current directory without write permission and with a corresponding RCS version file. The specific makefiles may go further:

- app: For the application with source code in the current directory or $(CODEDIR); deletes the associated dependency list and timestamp files, deletes the programmer library for the application, recursively deletes any tmplinst subdirectory (see *mkinst* (p180)), and deletes the executable for the application.
- aps: Deletes the dependency lists and timestamp files for all applications in this package, and deletes the executables for any applications with source code in system or programmer application subdirectories.
- chk: (none)
- doc: Deletes all .ps files generated from .latex, .texi, or .tex files, and recursively deletes all directories generated by *latex2html* from .latex sources.
- ftn: Deletes the programmer's private FORTRAN library for the package.
- imp: Deletes the dependency lists and timestamp files for the package/module, and deletes the programmer's private C++ library for the package.
- ins: Deletes the executables for any utility programs with source code in the current directory or $(CODEDIR).
- pkg: (none)
- scr: (none)
- top: (none)

  - `tst`: For all test programs with source code in the current directory or `$(CODEDIR)`; deletes the associated dependency lists and timestamp files, deletes the programmer library, recursively deletes any `tmplinst` subdirectory (see *mkinst* (p180)), and deletes the executables.

- `cleanest` : (general/specific)
  This target is defined generally in `makedefs` with a dependency which may be defined in the specific makefiles as the target of a rule to delete additional files. Generally this target just invokes the `cleaner` target. The specific makefiles may go further:

  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: (none)
  - `ftn`: (none)
  - `imp`: Deletes *all* of the programmer's dependency lists and timestamp files in `$(PGMRAUXD)`, deletes *all* object libraries in `$(PGMRLIBD)`, recursively deletes the programmer's template repository `$(PGMRPTRD)`, and recursively deletes any `tmplinst` subdirectory (see *mkinst* (p180)).
  - `ins`: (none)
  - `pkg`: (none)
  - `scr`: (none)
  - `top`: (none)
  - `tst`: (none)

  Note that since this target has the potential to delete much more than was possibly desired by the programmer it is advisable to do a dry run using

  ```
  gmake -n cleanest
  ```

  before using it.

- `%.lcc` : (specific, pattern rule)
  Preprocess a `.l` file using `$(LEX)` and leave the resulting `.lcc` file in `$(PGMRINCD)`.

  - `imp`: Preprocess the `.l` file associated (probably) with a class implementation.

- `%.ycc` : (specific, pattern rule)
  Preprocess a `.y` file using `$(BISON)` and leave the resulting `.ycc` file in `$(PGMRINCD)`.

  - `imp`: Preprocess the `.y` file associated (probably) with a class implementation.

- `%.d` : (specific, pattern rule)
  List the dependencies of a source file. A side effect of this is to update the dependency list for the particular source file if necessary. The contents of the programmer dependency list may differ from that for a system compilation of the same source file if the programmer has private versions of the source file or any of the relevant header files.

  - `app`: List the dependencies of an application.
  - `imp`: List the dependencies of a class implementation file.
  - `tst`: List the dependencies of a test program.

- `%.i` : (specific, pattern rule)
  Apply the `C` preprocessor `$(CPP)` to a source file in the current directory or `$(CODEDIR)` and put the resulting `.i` file in `$(PGMRLIBD)`.

  - `app`: Preprocess the `.cc` file for an application. The include path will be augmented with current directory (i.e. `-I.`) and of the corresponding system directory, `$(CODEDIR)`, in that order. This allows applications to maintain their own include files.
  - `imp`: Preprocess the `.cc` file for a class implementation.
  - `tst`: Preprocess the `.cc` file for a test program. The include path will be augmented with current directory (i.e. `-I.`) and of the corresponding system directory, `$(CODEDIR)`, in that order. This allows test programs to maintain their own include files.

- `%.o` : (specific, pattern rule)
  Produce an object module from the corresponding source file in the current directory or `$(CODEDIR)` and put it in `$(PGMRLIBD)`. Debug compilations are done by default, but optimized compilations can be selected via the `OPT` variable or alternate programmer compilation flags (see 11.3).

  - `app`: Compile the `.cc` file for an application into an object module and put it in `$(PGMRLIBD)`. The include path is augmented with current directory (i.e. `-I.`) and of the corresponding system directory, `$(CODEDIR)`, in that order. This allows applications to maintain their own include files.
  - `ftn`: Compile the `.f` file for a FORTRAN subroutine into an object module and put it in `$(PGMRLIBD)`.
  - `imp`: Compile the `.cc` file for a C++ class implementation into an object module and put it in `$(PGMRLIBD)`.
  - `ins`: Compile the `.c` file for a utility program into an object module and leave it in the current directory.
  - `tst`: Compile the `.cc` file for a test program into an object module and put it in `$(PGMRLIBD)`. The include path is augmented with current directory (i.e. `-I.`) and of the corresponding system directory, `$(CODEDIR)`, in that order. This allows test programs to maintain their own include files.

- `mylib(%.o)` : (specific, pattern rule)
  Compile an object module using the `%.o` pattern rule and insert it into a private library in `$(PGMRLIBD)`, then apply `$(RANLIB)` to the library.

  - `ftn`: Compile the `.f` file for a FORTRAN subroutine and insert it into a private FORTRAN library.
  - `imp`: Compile the `.cc` file for a C++ class implementation and insert it into a private C++ library.

- `%` : (specific, catch-all pattern rule or static pattern rule)
  Compile an object module or executable.
  - `app`: Catch-all pattern rule to compile an executable. Debug compilations are done by default, but optimized compilations can be selected via the `OPT` variable or alternate programmer compilation flags (see `makedefs` (p155)).

    The include path will be augmented with current directory (i.e. `-I.`) and of the corresponding system directory, `$(CODEDIR)`, in that order. This allows an application to maintain a set of include files specific to it.

    The executable will be linked against the library list specified by `$(PGMRLIBS)` in which private programmer libraries are listed before system libraries (see above).

    The executable will be left in `$(PGMRBIND)`.
  - `doc`: Catch-all pattern rule to invoke ***latex2html*** on the corresponding `.latex` source file.
  - `ftn`: Catch-all pattern rule to compile a FORTRAN subroutine if the corresponding library module is out-of-date. The resultant object module is left in the `$(PGMRLIBD)` directory and is not inserted into the library. Refer to the entry for the `imp` makefile.
  - `imp`: Catch-all pattern rule to compile a class implementation if the corresponding library module is out-of-date. The resultant object module is left in the `$(PGMRLIBD)` directory and is not inserted into the library.

    This rule may be used in conjunction with `update_mylib` to update several library modules in one invokation of ***gmake*** (p154) without updating the library for each one. For example,

    ```
    gmake Bloggs Boggle update_mylib
    ```

    Note that this is not the same as

    ```
    gmake Bloggs.o Boggle.o update_mylib
    ```

    Which may cause the object modules to be compiled even if the library modules are up-to-date with respect to their sources.
  - `ins`: Static pattern rule on `$(ALLEXES)` to compile a utility program leaving the executable in the current directory.
  - `tst`: Catch-all pattern rule to compile an executable. Debug compilations are done by default, but optimized compilations can be selected via the `OPT` variable or alternate programmer compilation flags (see `makedefs` (p155)).

    The executable will be linked against the library list specified by `$(PGMRLIBS)` in which private programmer libraries are listed before system libraries (see above).

    The executable will be left in `$(PGMRBIND)`.

- `depend` : (specific)
  Determine dependencies for all sources in the current directory.
  - `app`: Determine dependencies for the application.
  - `aps`: Determine dependencies for all applications which have their main `.cc` file in the programmer's workspace, `$(PGMREXES)`.

  - `imp`: Determine dependencies for all class implementations with sources in the current directory, `$(PGMRIMPS)`.
  - `tst`: Determine dependencies for all test programs with sources in the current directory, `$(PGMREXES)`.

- `dependencies` : (specific, recursive)
  Determine dependencies for all sources in the current directory and in `$(CODEDIR)`.
  - `aps`: Determine programmer dependencies for all programmer and system applications in this package, `$(ALLEXES)`. This comprises all applications which have their main `.cc` file in the programmer's workspace, `$(PGMREXES)`, and in the corresponding system directory, `$(AIPSEXES)`.
  - `imp`: Determine dependencies for all class implementations in the current directory and in `$(CODEDIR)` and likewise for all subdirectories.
  - `tst`: Determine dependencies for all test programs in the current directory and in `$(CODEDIR)`.

- `exorcise` : (specific)
  Delete stale entries from dependency lists and programmer libraries after a source file has been removed from the current directory (e.g. via checkin). Recurses into any `tmplinst` subdirectory to do the same thing there.
  - `app`: Cleans up the dependency list and programmer library for the application.
  - `imp`: Cleans up the dependency list for the software module and the programmer library for the package.
  - `tst`: Cleans up the dependency list and programmer library for the test program.

- `mylib` : (specific)
  Compile all sources in the current directory and in `$(CODEDIR)` to build a complete private object library.
  - `app`: Instantiate do-it-yourself templates (see ***mkinst*** (p180)). If a semaphore file called `MyTemplatesOnly` exists in the programmer's directory then system templates will be excluded from the compilation.
  - `ftn`: Build a private FORTRAN library.
  - `imp`: Build a private C++ library.
  - `tst`: Instantiate do-it-yourself templates (see ***mkinst*** (p180)). If a semaphore file called `MyTemplatesOnly` exists in the programmer's directory then system templates will be excluded from the compilation.

- `mylibs` : (specific)
  Invoke the `mylib` rule in the current directory and all subdirectories.
  - `ftn`: Invoke `mylib` in the current directory and all subdirectories.
  - `imp`: Invoke `mylib` in the current directory and all subdirectories.

- `tmplinst` : (specific)
  Invoke ***mkinst*** (p180) to generate do-it-yourself template instantiation files. If there is no `templates` file in the current directory then ***mkinst*** is invoked with input from `/dev/null` in order to delete any stale template instantiation files.

  - `app`: Generate do-it-yourself template instantiation files for an application.
  - `imp`: Generate do-it-yourself template instantiation files for a class implementation module.
  - `tst`: Generate do-it-yourself template instantiation files for an test program.

- `inst` : (specific)
  Invoke the `tmplinst` rule to generate do-it-yourself template instantiation files and then invoke the `mylib` rule in the `tmplinst` subdirectory to compile and insert them into an object library.

  - `app`: Generate and compile do-it-yourself templates for an application.
  - `imp`: Generate and compile do-it-yourself templates for a class implementation module.
  - `tst`: Generate and compile do-it-yourself templates for a test program(s).

- `update_mylib` : (specific)
  Insert all object modules into a private library and then apply `$(RANLIB)` to it. The object modules and library reside in `$(PGMRLIBD)`. The object modules are deleted after the library has been updated.

  - `app`: Update a private object library associated with an application.
  - `ftn`: Update a private FORTRAN library.
  - `imp`: Update a private C++ library.
  - `tst`: Update a private object library associated with a test program.

- `%.dvi` : (specific, pattern rule)
  Compile a `.dvi` file.

  - `doc`: Compile the `.dvi` file from the `.texi`, `.latex`, or `.tex` file (in that order of preference) in the current directory or `$(CODEDIR)` and leave it in the current directory.

    Included files will be searched for in the corresponding `.dir` subdirectory of the current directory and of `$(CODEDIR)` in that order. Any LaTeX-related `.bib` bibliography files contained in these directories will be processed by BibTeX.

- `%.ps` : (specific, pattern rule)
  Compile a `.ps` PostScript file.

  - `doc`: Compile a `.ps` PostScript file from the corresponding `.dvi` file in the current directory. If necessary the `.dvi` file will be generated automatically by *gmake* (p154) by rule-chaining the `%.dvi` pattern rule.

- `doc` : (specific)
  Compile all printable documents.

  - `doc`: Compile all `*.texi`, `*.latex` and `*.tex` files in the current directory into `*.ps` PostScript files and leave them in the current directory.

Notes

- The implement makefile includes the system dependency lists and so may start slowly as it checks to see whether the list is up-to-date (regardless of whether the particular target uses it or not). Inclusion of the dependency list can be circumvented by setting the `NODEP` variable (to anything). This causes the makefile to start considerably faster.

  `NODEP` is set automatically if ***gmake*** (p154) is invoked from a directory which does not reside under `$AIPSROOT`, that is, a programmer invokation. However, system-oriented targets may also be invoked from a programmer directory, and this would cause the dependency analysis to be circumvented, and the dependency list to be ignored. This may be legitimate if the target invoked does not actually use the dependency list, for example `chkout`, `cleancode` or `cleansys`. In fact, the code distribution system sets `NODEP` explicitly when it invokes the top-level makefile for these recursive targets for the system (***inhale*** (p78)). However, `NODEP` must not be set for an invokation of the implement makefile for a target which does use the dependency list. A proper resolution of this problem would require the facility for a makefile to examine its target list, but this is not currently possible.

- If the AIPS++ RCS source code repositories are present in an installation all AIPS++ makefiles access them via a symbolic link `$AIPSROOT/rcs` which is usually set to point to `slave`. Therefore, the slave RCS repositories are normally the ones consulted. However, it is possible to reset the `rcs` "switch" to `master` to cause ***gmake*** (p154) to checkout and/or rebuild AIPS++ from the `master` repositories. ***exhale*** (p73) uses this mechanism when constructing a new base release. However, it requires good bandwidth to the master and is only feasible in Socorro.

Examples

The `mylib` target for the implement makefile rebuilds a programmer's private library in its entirety, including all object modules generated from sources in `$(CODEDIR)`. To rebuild only the modules with sources in the programmer's directory the following would suffice:

```
cd $HOME/aips++/code/aips/implement
gmake
```

This is the recommended way of updating a private libarary.

See also

The GNU `Make` manual.
The GNU manual page for ***gmake***.
The unix manual page for ***ranlib***(1).
AIPS++ variable names (§1.2, p8).
gmake (p154), GNU make.
`makedefs` (p155), AIPS++ makefile definitions.
mkinst (p180), Generate template instantiation files from a list.
updatelib (p183), update an AIPS++ object library.

11.5 *depstat*

Count the number of header files used by AIPS++ class implementation files.

Synopsis

depstat `[dependency list]`

Description

depstat is a *sed*/*awk* filter which reads a dependency list and counts the number of AIPS++ header files on which each AIPS++ implementation file depends.

System dependency lists have names of the form `*.list` and reside in the `$(ARCHAUXD)` directory. Programmer dependency lists reside in `$(PGMRAUXD)` if that exists, otherwise they are left with the programmer's source code. Refer to §11.4, p199 and §11.3, p196.

If no dependency list is specified on the command line, input is taken from `stdin`.

Options

None.

Examples

See also

AIPS++ variable names (§1.2, p8).
Section §11.3, p196, Code development `makedefs` conventions.
Section §11.4, p199, Code development makefile rules.

Author

Original: 1994/02/10 by Mark Calabretta, ATNF

11.6 *ldmap*

Determine dependencies between object modules from a namelist.

Synopsis

```
ldmap
   [-1]
   [-h #]
   [-m | -M]
   [-o | -O object]
   [-r]
   [-s file | directory]
   [-t]
   [namelist [namelist ...]]
```

Description

ldmap reads through a file containing the output produced by *nm* and determines the dependency hierarchy of `.o` files. That is, if `X` is an undefined symbol in `a.o`, and `X` is defined in `b.o`, then `a.o` depends on `b.o`.

There is also an option to reverse the dependency analysis so that instead of listing all the object modules *required by* a particular object module *ldmap* lists all of the object modules *which require* a particular object module.

The name list file may be specified as "`-`" whence *ldmap* will read it from `stdin`.

The analysis is done in two stages. First the namelist is converted to a list of dependencies of the form

```
a.o b.o
```

meaning that `a.o` requires `b.o`. If the *-m* option is specified the output looks like

```
a.o usym b.o
```

where `usym` is the undefined symbol in `a.o` which would cause the linker to load `b.o`. If the *-M* option is specified the output also includes lines such as

```
a.o dsym a.o
```

where `dsym` is a symbol defined in `a.o`.

Unresolved symbols may be included in the output via the *-n* option:

```
a.o (usym)
```

The stage 2 analysis consists of iterating through the dependency list to find all object modules required by each particular object module. By default, all object modules in the dependency list are processed, however the analysis may be done for a set of object modules via the *-o* option.

Since the stage 1 analysis is time-consuming, the intermediate results can be saved for later use in either or both of two forms. In packed form, the stage 1 results are stored in a single file. In the unpacked form, which is the most efficient for the stage 2 analysis, the results for each object module are stored in a separate file in a subdirectory.

Options

-1 Quit after the stage 1 analysis.

-h `#` Set a limit to the depth of the dependency hierarchy in the stage 2 analysis.

-m | *M* Produce a stage 1 analysis with more information than is needed for stage 2.

-m includes symbol names in the output of the stage 1 analysis, and *-M* also includes symbols defined in each object module.

The *-m* and *-M* options are meant to be used with the *-p* option to produce a file for visual inspection. The extra information is thrown away when the stage 1 analysis is unpacked.

-n Include unresolved symbols in the stage 1 analysis. They appear in the stage 1 output in parentheses.

-o `object` | *-O* `object` Determine dependencies for the specified object module only. Multiple object modules may be specified in a single *-o*, or multiple *-o* options may be specified. Either way the analysis is performed for the group as a whole.

If *-O* is specified the stage 1 analysis will be unpacked, otherwise the packed analysis will be used unless the unpacked analysis is already available.

-p `file` | *-P* `directory` Preserve the result of the stage 1 analysis.

-p saves the packed result in the specified file.

-P saves the unpacked result in the specified directory.

If the file and/or directory already exist they will be deleted. Both options may be specified together.

-r Reverse the direction of the dependency analysis. In the stage 1 analysis the namelist is converted to a list of dependencies of the form

```
a.o b.o
```

meaning that `a.o` is required by `b.o`. The stage 2 analysis is unaffected. Use of the *-r* option precludes the use of the *-n* and *-v* options.

-s `file` | *-S* `directory` Skip stage 1 of the analysis.

-s specifies the name of a file containing the stage 1 analysis from a previous invokation of `ldmap -p`, or the name of a directory containing the unpacked stage 1 analysis from a previous invokation of `ldmap -P`.

In either case the namefile need not be specified.

A *-s* option may not be combined with stage 1 analysis options except for the combination `-s <file> -P <directory>`.

-t Print timing information.

-v `file` A file containing the output of `ar tv` used for determining sizes. Multiple *-v* options may be specified.

Notes

- *ldmap* assumes the namelist is in a form similar to that produced by the *nm* provided with SunOS 5.5. Other versions of *nm*, even those in earlier versions of SunOS, may produce namelists in an incompatible format.
- The output of ldmap may be piped through *c++filt* to demangle C++ names.

Diagnostics

Status return values
`0`: success
`1`: initialization error

Examples

The following determines all dependencies for the `gtable` application.

```
cd $HOME/aips++/code/aips/apps/gtable/
gmake gtable.o
cd $HOME/aips++/sun4sol_gnu/bindbg
nm gtable.o > gtable.nm
nm /aips++/sun4sol_gnu/lib/libaips.a > libaips.nm
ldmap -n -o gtable.o gtable.nm libaips.nm
```

This produces output such as

```
gtable.o
   Error2.o
   Excp2.o
   GlishArray.o
   GlishEvent.o
   GlishRecord.o
   String.o
      ArrayPosIter.o
      Array_1020.o
      Array_1030.o
      Array_1050.o
```

```
        .
        .
        .
              StreamLogSink.o
              SymLink.o
              Time.o
                 ByteIO.o
                 ObjectID.o
                    Vector_1100.o
                       VectorRtti_1180.o
Count: 235
```

The levels of indentation are indicative of each cycle of the recursion. Note that the above list contains object modules derived from do-it-yourself template instantiation (see *mkinst* (p180)).

Using `libaips.nm` generated above, the following saves a reversed stage 1 analysis of `libaips.a` in unpacked form for later use and then quits.

```
ldmap -1 -P libaips.1ru -r libaips.nm
```

With most of the hard work done, the saved stage 1 analysis may then be used repeatedly to quickly determine all object modules in `libaips.a` which depend on (i.e. contain symbols resolved by) a specified module. For example `DeconvTool.o`

```
ldmap -s libaips.1ru -o DeconvTool.o
```

resulting in the following (unusually brief) output:

```
DeconvTool.o
   CCleanTool.o
   CleanTool.o
   FTCleanTool.o
      HCleanTool.o
      SDICleanTool.o
Count: 6
```

See also

The unix manual page for *nm*(1).
mkinst (p180), Generate template instantiation files from a list.

Author

Original: 1996/05/06 by Mark Calabretta, ATNF.

11.7 Template management

11.7.1 *reident*

Remake the idents in a templates list.

Synopsis

reident [-*z*] [-*t*] [file [file ...]]

Description

reident reads a template file (`templates` by default, see ***mkinst*** (p180)), sort/uniqs it on the name of the class implementation `.cc` file (second field), and replaces the idents on some or all lines so that each combination of ident and `.cc` file is unique.

reident also converts each entry in the input to a canonical form with respect to the use of spacing and `typedef`'d variables so that trivial duplications are automatically eliminated, and it warns of any non-trivial deviation from the canonical forms.

reident does not check for duplicate template definitions. The ***duplicates*** (p216) program is available for that purpose.

reident also converts each entry in the input to a formatted multi-line output that can be read by ***mkinst*** (p180).

If multiple files are specified they will be merged together. One file may be specified as "`-`" to denote input from `stdin`. Output is written to `templates` or to `stdout` – if the input came from `templates` it will be overwritten.

Missing idents (the preferred way of adding template definitions), idents specified as less than `1000` (another safe way to add new templates) and the second and subsequent occurrence of duplicate idents are replaced by a unique value for each `.cc` file. If the ***-z*** option is specified, all idents are replaced with an ascending sequence starting at `1000` for each `.cc` file.

The idents are incremented by `10` to allow manual insertion of new template declarations in the correct sequence.

Comments in the templates file (lines beginning with "`#`") will either stay at the top of the file, or stay tuned to the template definition line(s) they occur at.

Options

-z Replace all idents with an ascending sequence restarting at `1000` for each `.cc` file.

-t Write the output to the terminal (`stdout`) in stead of `./templates`.

Notes

- Changing the ident for a template declaration invalidates any pre-existing template instantiation `.cc` file for that template. However, since the idents restart for each class implementation .cc file, ident changes will be quarantined to the templates based on it.

Diagnostics

Status return values
0: success
1: file not found

Examples

To merge a private templates list into the system's list for the aips package and at the same time completely refresh the idents

```
cd $HOME/aips++/code/aips/implement/_ReposFiller
ao -l templates
reident -z templates myTemplates
ai templates
```

To check a newly created templates list and check it for duplicates internally and against the trial repository:

```
reident -t | duplicates -r trial -
```

If seldom used the *-z* option has the potential to cause a great deal of recompilation because of accumulated changes. However, if used regularly the incremental cost should be modest. Also, all care should be taken that templates included are not duplicated (see ***duplicates*** (p216)).

See also

mkinst (p180), Generate template instantiation files from a list.
duplicates (p216), Checks for duplicate entries in templates lists across the system.
unused (p218), Checks for usage of templates in binary modules (executables or object modules).
used (p221), find any undefined classes and global methods.

Author

Original: 1996/04/16 by Mark Calabretta, ATNF.

11.7.2 *duplicates*

Look for duplicate entries in AIPS++ `templates` instantiation files

Synopsis

duplicates [-*p* *package*|*all*] [-*r* *package*|*all*] [-*u* *package*|*all*] ... [templates file] ...

Description

duplicates is a `C++` executable program which looks for duplicate entries in AIPS++ `templates` files. This is useful for finding redundant templates entries, and for moving duplicate entries into a common `templates` file.

One entry in the `template file list` may be specified as `-` to indicate standard input.

If no list of files is provided, the list used is the combination of:

1. A file named `templates` in the current working directory if any; and
2. All files named `templates` under the `$AIPSCODE` tree, unless switches restrict the search to specified packages and/or repositories and/or non-repositories.

Duplicates found in the first (or the default templates file) are indicated by asterisks next to the line number.

The output can be restricted to duplicates that are only present in a package repository file and/or in a single file.

Options

Options are provided to limit the number of templates files that will be included in the search. Check with:

```
duplicates -h
```

to find the currently available switches.

-*h* Give some useage information

-*s* Restrict the output ('s' for 'system') to duplicates found in a single file or in `_ReposFiller` files.

-*p* *package*|*all* Restrict tests to templates files in the specified package (or all packages)

-*r* *package*|*all* Restrict tests to the repository templates files in the specified package (or all packages)

-*u* *package*|*all* Restrict tests to the non-repository templates files in the specified package (or all packages)

Examples

```
duplicates
duplicates templates /aips++/code/aips/implement/_ReposFiller/templates
reident -t | duplicates -r trial -
```

The first line finds all duplicates in `./templates` and all templates files checked into the system. The second example finds duplicates only in the listed files. The third line reformats and checks `./templates`, and then checks for duplicates in this file and in `trial/implement/_ReposFiller/templates`.

See also

AIPS++ do-it-yourself template instantiation (***mkinst*** (p180)).
reident (p214), reformat in canonical form the template instantiation file.
unused (p218), check usage of templates in binary executables and object modules.
used (p221), find any undefined classes and global methods.

Author

Original: 1997/06/09 by Brian Glendenning, NRAO

11.7.3 *unused*

Check for unused (or used) template object modules in binaries and/or object modules and/or libraries.

Synopsis

unused [*-z* | *-nz* | *-a* | *-[0-9]...*] [*-p* | *-r* | *-u* package] ... [-t templates1 [templates2 ...]] *-f* | *-b* file1 [file2 ...] [*-i* list1 [list2 ...]]

Description

unused reads binary files (either executables, object modules or object libraries) specified by the *-b* switch or the output of the *g++filt* filtered output of the *nm* or *gnm* program of such files as specified by the *-f* switch. I.e. files that are to be specified following an *-f* switch should have the following pre-processing:

```
[g]nm binary.file | g++filt >! file.f
```

(Note that internally ***unused*** uses the *gnm* and *g++filt* synonyms. If they are not available on your system, the full path of these programs (or for the *nm* program) can be specified in the `aipsrc` file variables `unused.file.nm` and `unused.file.gfilt`.)

The other part of the input is a set of templates files. They can be specified as individual templates files (following the *-t* switch), or as a set of templates files in a package (like *aips*, *trial* or *all* to indicate all packages).

The templates in the specified templates files (including those specified with the *-i* switch) will be checked against the templates defined in the binary files . Depending on the show switches (*-a*, *-z*, *-nz*, *-number*) the output will contain templates used or unused in the specified binary files.

Options

-z Show the unused templates only (this is the default if no show switch is given)

-nz Show the used templates only (if the output is saved to a file, it can be used in a subsequent run of the ***unused*** program with the *-i* switch in stead of the original binary files)

-a Show all the templates used in the binaries (i.e. both the used and unused ones)

-number Show only the templates that are used in at least the number of binary files specified. I.e. if the switch given is *-2*, and two binary files are given as input, only templates occurring in both of these will be shown. If only one binary file was given nothing will be shown; if three binary files were specified all templates used in at least two of them will be shown.

-p package Use all templates files found in the specified package (the package name may be "all")

-r package Use all templates files in the repository directory of the specified package (or "all")

-u package Use all user templates files in the package (i.e. excluding the ones in the repositories)

-t file [file ...] Use the explicitly specified templates files

-i file [file ...] Use the specified files as a source of templates. The files must be the ouput of an earlier run of the unused program (e.g. with the *-nz* switch).

-b|-f file [file ...] Specify the binary files (*-b*) to be checked. The files can be executables, object libraries or object modules (i.e. files that can be handled by some *nm* program). If the *-f* switch is used in stead of the *-b* switch, the binary files are supposed to have been filtered already by a *nm* program and the *g++filt* program.

Notes

- The output of the program can be used as input to the *duplicates* (p216) program.

Diagnostics

Status return values
 0: success
 1: no files found

Examples

To show all the templates specified in the `Tables/test/templates` file that are used in a demonstration program *dM30*:

```
   unused -a ~/aips++/sun4sol_gnu/bindbg/dM30 -b `which dM30`

Using [/aips++/code/aips/implement/Tables/test/templates]
176 templates in 1 files
176 templates in 1 files after 0 inclusion files
Checked against 11259 names in 1 name files
# Template usage:
    0: AiPs_hidden_type_(Array<ExampleDesc> *)
    0: AiPs_hidden_type_(Array<RetypedArrayEx1> *)
    0: AiPs_hidden_type_(Array<RetypedArrayEx2> *)
    0: AiPs_hidden_type_(Array<VSCExample> *)
.......
    0: valDataTypeId(RetypedArrayEx2 const *)
    0: valDataTypeId(VSCExample const *)
176 templates shown
```

In this case all the templates in the templates file have a count zero (none used in the binary). The same output would have been generated with the (default) *-z* switch, while no templates would have been shown if the *-nz* switch had been used.

See also

mkinst (p180), Generate template instantiation files from a list.
duplicates (p216), Checks for duplicate entries in `templates` lists across the system.

Author

Original: 1997/07/16 by Wim Brouw, ATNF.

11.7.4 *used*

Check for undefined classes and global functions in object libraries and object modules.

Synopsis

```
used [-ng] [-nc] [-tc] [[-f|-b] file1 [file2 ...]] ...
```

Description

used reads binary files (either object modules or object libraries) specified by the *-b* switch (default) and/or the output of the *g++filt* filtered output of the *nm* or *gnm* program of such files if specified by the *-f* switch. I.e. files that are to be specified following an *-f* switch should have the following pre-processing:

```
[g]nm binary.file | g++filt >! file.f
```

(Note that internally *used* uses the *gnm* and *g++filt* synonyms. If they are not available on your system, the full path of these programs (or for the *nm* program) can be specified in the `aipsrc` file variables `unused.file.nm` and `unused.file.gfilt`.)

The first file specified is checked for any undefined global names. These undefined names are then checked against any definitions in all files given. Finally the resulted list of undefined names is output to stdout. The standard checking is for all global names. The *-ng -nc* and *-tc* switches can be used to limit the output.

Options

-nc Show no classes (i.e. only the global functions)

-ng Show no global functions (i.e. only classes)

-tc Show for classes only templated classes

-b|-f file [file ...] Specify the binary files (*-b*), also the default, to be checked. The files can be executables, object libraries or object modules (i.e. files that can be handled by some *nm* program). If the *-f* switch is used in stead of the (default) *-b* switch, the binary files are supposed to have been filtered already by an *nm* program and the *g++filt* program.

Notes

- The output of the program can be used as input to the *duplicates* (p216) program.

Diagnostics

Status return values
`0`: success
`1`: no files found

Examples

To show the missing templated classes used in the given file:

```
   used -ng -tc  ~/aips++/sun4sol_gnu/libdbg/libaips-Measures-test.a

Using [/u/wbrouw/aips++/sun4sol_gnu/libdbg/libaips-Measures-test.a]
Checked 7 undefineds against 2 defineds
# Undefined templated classes
    1: Array<Bool>
   18: Array<Quantum<Double> >
    7: Array<Double>
   14: Quantum<Vector<Double> >
   18: Quantum<Double>
    6: Vector<Double>
6 undefined objects/global functions shown
```

To check if the modules in libtrial.a have still some undefined references:

```
   used /aips++/weekly/sun4sol_egcs/lib/{libtrial,libaips}.a
```

The counts shown are the number of references to the class or global function found.

See also

mkinst (p180), Generate template instantiation files from a list.
unused (p218), Checks for usage of templates in binary modules (executables or object modules).
duplicates (p216), Checks for duplicate entries in `templates` lists across the system.

Author

Original: 1997/09/14 by Wim Brouw, ATNF.

11.8 UP scripts to change and move code

The UP scripts' main purpose is to either change occurrences of certain patterns in the AIPS++ code (*UPchange* (p234)), or to move code files from one directory to another, and update all references to the files (*UPmove* (p235)). A list of files to be processed is generated in all cases most easily by the *UPfind* (p227) script (possibly with manual editing of the list).

In addition some general scripts to aid the system adiministration (*UPlist* (p229), *UPreident* (p243), *UPdup* (p241), *UPlock* (p231), *UPtmpl* (p240), *UPload* (p239), *UPcasa* (p233)) exist.

Information is exchanged between the different scripts by files. These files are created in the current working directory, and all have the same name, but different extensions. The name is the code used to bind different calls together. To change all files that contain the word *cat* into *dog*, the following sequence would do that (strcat being the code name):
Find the list of files containing the word cat:

```
UPfind strcat
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/nfs/aips++/weekly]:
Specify egrep expression(s) to find files
Examples: /Measures/((MUString)|(MVAngle)|(MVTime)|(RotM)|Q|(Un))
      or: [(][^(]*[*] *[)][ \t]*0
Multi-line expressions will be ORed
Find expression: [^a-zA-Z_0-9]cat[^a-zA-Z_0-9]
Find expression:
Specify egrep pattern for files to be removed from raw list found.
Example: [/]aipsview[/]
tmplinst files will always be removed
Remove glish files? (y|n) [y]:
Remove pattern:
Starting find...
9 files found in strcat.raw
Starting selected removal...
9 files left in strcat.found
```

The files selected, and the lines that were used in the selection, can be checked by

```
UPlist strcat
Getting info...
9 files found in strcat.found
Starting list...
47 lines in strcat.list
```

The change is then made by:

```
UPchange strcat
Getting info...
```

```
Specify the sed patterns used to change the contents of the files
in the files in the strcat.found list.
Example: s#/Measures/MV#/Quanta/MV#g
         s#[.]ac[(][)]##g
         s#at_cc[(]#(#g
         s#\([^a-zA-Z_]\)assert\([^a-zA-Z_]\)#\1assure\2#g
Change pattern: s#\([^a-zA-Z_0-9]\)cat\([^a-zA-Z_0-9]\)#\1dog\2#g
Change pattern:
Specify the message lines for the RCS log
Example: Moved includes from Quanta to Measures
Text line: Changed the cat into the dog
Text line:
9 files will be changed and checked in/out
./atnf/apps/atcafiller/DOatcafiller.cc started
```

...

Information exchange files produced at this stage:

```
strcat.def
strcat.done
strcat.error
strcat.found
strcat.list
strcat.raw
strcat.rcs
strcat.rmpat
strcat.sed
strcat.spat
```

The scripts are written assuming tcsh is available at the computer used, and can be accessed by */usr/bin/env*. The script may have to be changed if this is not the case. Also, the ***UPmove*** (p235) script uses the *amv* command, only available at the master site, and can only be run from there.
Note that the example outputs are only indicative, and could be different in details.

11.8.1 *UP**

General description of the ***UP*** scripts.

Synopsis

*UP** [-*<switches>*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type name>

Description

Each script has the *¡type name¿* as an argument. This type name is the file name (with varying extensions) that binds the different scripts that have to cooperate together. If no type name is

given, the user is prompted for one, or one that has been used already that day will be used. It is advisable to always provide a name.
Switches can be given either separately, or bunched together.
The output of the scripts is given in files with the type name as main name.

The type name act as a memory from invocation to invocation. In general once information is provided for a specific type, that information will not be asked for again (unless one of the clearing switches is present, or the communication files are deleted). Most information is script specific, but some genarl information is saved in the general definition file *.def*.

Options

All scripts accept the following switches; while all the scripts actually changing system information have also a *-t* test switch, which shows what would have happened had the script run for real.

-h Give some useage information

-c Clear all defining information files for the *<type name>* invocation of the script, and re-ask it.

-r Clear and re-ask only the general definitions for *<type name>*

-u Clear and re-ask only the specific definitions for *<type name>*

-d Debug mode. Some more output could be generated by some scripts. No extensive implementation present.

-n Will only generate the information needed to run the script, without executing the real code

-t Available for some scripts only. When present will execute (if no *-n* present) the real code, but will not change the system. In general the actual change is shown in a *.diff* file.

-Q A switch, mainly for internal use, that indicates that the script will run quiet, and only notify of errors, and asking questions. No informational messages (like examples of input) are given.

Examples

Help information:

```
UPfind -h
Usage:   UPfind [-f] [-g] [-c] [-r] [-u] [-d] [-n] <typ>
  Get all files that contain data according to an egrep rule.
  The found files can be used in other UP scripts like UPlist,
  UPchange etc. The <typ> is an id acting (by virtue of files)
  as a communication between the different UP scripts.
  All .cc, .h and templates files in the code tree are searched.
Uses:    <typ>.def, <typ>.spat, <typ>.rmpat
Creates: <typ>.raw, <typ>.found
```

```
  -g use all .g files rather than the C++ files
  -f use a file pattern to search for files, not their contents
  -c clear all defining information files for this script
  -r clear only the general defining information in .def
  -u clear all but the general defining information in .def
  -d give some debug information (mostly not implemented)
  -n run script only to get defining information, without executing
  -Q run quietly
  -F force acceptance of normally asked for data
  -h this information

The following UP scripts are available:
    UPfind    [-f] [-g]      [-c] [-r] [-u] [-d] [-n] <typ>
    UPlist    [-l]           [-c] [-r] [-u] [-d] [-n] <typ>
    UPlock    [-a] [-l]      [-c] [-r] [-u] [-d] [-n] <typ>
    UPchange  [-t]           [-c] [-r] [-u] [-d] [-n] <typ>
    UPmove    [-i] [-t]      [-c] [-r] [-u] [-d] [-n] <typ>
    UPload                   [-c] [-r] [-u] [-d] [-n] <typ>
    UPtmpl                   [-c] [-r] [-u] [-d] [-n] <typ>
    UPdup     [-s]           [-c] [-r] [-u] [-d] [-n] <typ>
    UPreident [-a] [-t] [-z] [-c] [-r] [-u] [-d] [-n] <typ>
    UPcasa
```

Providing only the information to run (and clear all information first):

```
UPfind -nc strcat
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/nfs/aips++/weekly]:
Specify egrep expression(s) to find files
Examples: /Measures/((MUString)|(MVAngle)|(MVTime)|(RotM)|Q|(Un))
      or: [(][^(]*[*] *[)][ \t]*0
Multi-line expressions will be ORed
Find expression: [^a-zA-Z_0-9]cat[^a-zA-Z_0-9]
Find expression:
Specify egrep pattern for files to be removed from raw list found.
Example: [/]aipsview[/]
tmplinst files will always be removed
Remove glish files? (y|n) [y]:
Remove pattern:
```

Providing only the information to run (and clear general information first, and re-use other information):

```
UPfind -n -r strcat
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/nfs/aips++/weekly]:
```

Providing only the information to run (and re-use all information):

```
UPfind -n strcat
Getting info...
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.2 *UPfind*

Find files in system code tree. File list to be used in action *UP* scripts.

Synopsis

UPfind [*-g*] [*-f*] [*-c*] [*-r*] [*-u*] [*-d*] [*-n*] `<type>`

Description

The *UPfind* script finds all files in the AIPS++ code tree (normally in the .{*h,cc*} files, but see the *-g* flag) that contain the specified pattern. The files found will be stored in the `<type name>.raw` file. A filter is applied to leave files to be processed in the `<type>.found` file.

The information supplied in answer to questions asked will be stored as well, for the use in subsequent scripts with the same `<type>`.

Options

-g Look for files in `.g` rather than `.h`, `.cc` and `templates` files.

-f Generate the file list from a set of *file* patterns, rather than a set of *egrep* patterns. The *file* patterns should be rooted at *AIPSROOT/code*.

Examples

```
UPfind -h
Usage:   UPfind [-f] [-g] [-c] [-r] [-u] [-d] [-n] <typ>
  Get all files that contain data according to an egrep rule.
  The found files can be used in other UP scripts like UPlist,
  UPchange etc. The <typ> is an id acting (by virtue of files)
  as a communication between the different UP scripts.
  All .cc, .h and templates files in the code tree are searched.
Uses:    <typ>.def, <typ>.spat, <typ>.rmpat
Creates: <typ>.raw, <typ>.found
  -g use all .g files rather than the C++ files
  -f use a file pattern to search for files, not their contents
```

See the general introduction for an extensive example

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.3 *UPlist*

List the content of files found by the ***UPfind*** command.

Synopsis

UPlist [-*l*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPlist*** script lists all the files listed in the ***<typ>.found*** file. Only those lines are listed that contains the selection pattern from the UPfind command. The list is given in ***<type>.list***. The command can be used to check the selection of files before actually changing or moving files, but is also valid in its own right.

Options

-*l* Provide a line number in the file lists as well.

Examples

```
UPlist -h
Usage:   UPlist [-l] [-c] [-r] [-u] [-d] [-n] <typ>
  List all lines that have been found using an egrep rule in
  <typ>.spat. The files are given in <typ>.found.
  In general the pattern and files are listed using UPfind.
Uses:    <typ>.def, <typ>.spat, <typ>.found
Creates: <typ>.list
  -l will give the line numbers in the files
```

Using the find information from the general introduction, we get:

```
UPlist
Assuming <typ> as 'strcat'.
Getting info...
9 files found in strcat.found
Starting list...
47 lines in strcat.list

cat strcat.list
1000 ## ./atnf/apps/atcafiller/DOatcafiller.cc
1001 ## ./bima/apps/bimafiller/bimafiller.cc
1002 ## ./nrao/apps/gbtobsparser/GBTObsSymtab.h
1003 ## ./aips/implement/Utilities/test/tString.cc
1004 ## ./aips/implement/Utilities/String.cc
1005 ## ./trial/implement/MeasurementSets/MSConcat.cc
```

```
1006 ## ./trial/implement/MeasurementSets/MSFitsInput.cc
1007 ## ./trial/implement/MeasurementSets/FitsIDItoMS.cc
1008 ## ./trial/apps/aipsview/AvString.cc
1000 ::   Vector<String> cat(nCat);
1000 ::   cat(0)="FLAG_CMD";
1000 ::   cat(1)="ORIGINAL";
1000 ::   cat(2)="USER";
1000 ::   msc_p->flagCategory().rwKeywordSet().define("CATEGORY",cat);
1001 ::   Vector<String>  cat(nCat);
```

...

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.4 *UPlock*

Check and report if files are locked.

Synopsis

UPlock [-*a*] [-*l*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPlock*** script lists all the files that are locked in the master. Listed in ***<type>.lock*** are either the files in the ***<typ>.found*** file that are locked in the AIPS++ master tree, or all locked files. The command can be used to check the status of files before actually changing or moving files, but is also valid in its own right. If the command is run from a local installation, it is assumed that the user has an account at the aoc aips master suitable for ssh (*i.e.* if you are able to do ***rao***). If the account name differs from the local name, the `$AUID` environment variable has to be set with the remote user name (like for ***rao***).

Options

-a Show all locked files, rather than the selection based on the ***<type>.found*** list

-l Provide the locker and the revision level in addition to the file name.

Examples

```
UPlock -h
Usage:   UPlock [-a] [-l] [-c] [-r] [-u] [-d] [-n] <typ>
  Test for each locked file if it is in the list of files
  <typ>.found (as created with UPfind), and output the
  result in the file <typ>.locked.
Uses:    <typ>.def, <typ>.found
Creates: <typ>.locked
  -a display all locked files, rather than a selection based
     on <typ>.found
  -l show the locker and revision number as well
```

Get all locked files and their lockers:

```
  UPlock ct -la
  Getting info...
  Finding locked files...
  201 files in ct.locked

  cat ct.locked
  akemball 15.0 ./aips/implement/MeasurementSets.h
```

```
akemball 15.0 ./bima/apps/bimafiller/bimafiller.g
akemball 15.0 ./bima/apps/bimafiller/bimafiller.help
```

...

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.5 *UPcasa*

Return the list of basic casa library packages..

Synopsis

UPcasa

Description

The *UPcasa* script lists all the basic casa library packages in hierarchal order. The command is an easy way to get the hierarchal list of casa library packages.

Options

Examples

```
UPcasa
casa scimath tables measures fits lattices
coordinates components images ms msvis
calibration ionosphere flagging dish
simulators synthesis graphics tasking
```

Author

Original: 2004/10/15 by Wim Brouw, ATNF

11.8.6 *UPchange*

Change source files in AIPS++ master, using a list of files, probably created by `UPfind`.

Synopsis

UPchange [-*t*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPchange*** script changes all the files given in ***<type>.found***, using the editor pattern provided. Successful changes are recorded in ***<type>.done***, unsuccesful ones in ***<type>.error***. If the **-*t*** test switch is given, no real changes are made, but the differences are recorded in ***<type>.diff***.

Options

-*t* Do only a test run. Files are checked out and changed, but rather than checking the changed files in again, the differences created are recorded.

Examples

```
UPchange -h
Usage:   UPchange [-t] [-c] [-r] [-u] [-d] [-n] <typ>
  Update all files given in the <typ>.found list (which is
  normally created by UPfind). Updating means that files
  will be checked out, changed and checked in again.
  <typ>.done and <typ>.error will contain the files successfully
  done or not.
Uses:    <typ>.def, <typ>.found, <typ>.sed, <typ>.rcs
Creates: <typ>.done, <typ>.error, <typ>.diff
  -t will run a test mode, where files are checked out,
     changed, but not checked in again. A file <type>.diff will
     be created showing what would have been changed.
```

See the general introduction for an extensive example

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.7 *UPmove*

Move source files in AIPS++ master, using a list of files, probably created by `UPfind`. The files will be moved from one path to another, updating all files referencing these files as well.

Synopsis

UPmove [-*i*] [-*t*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPmove*** moves files in the AIPS++ code tree from one place (and/or name) to another. At the same time it will change all references to that file in the system as well. The output of the script is a list of files that were successfully changed in ***<type>.done***, and in ***<type>.error*** for unsuccessful ones. If the *-t* test switch is given, the ***<type>.diff*** will give differences that would have been made when the file is run for real.
If files are actually moved, the script must be run from the aoc master, since `amv` is involved (see ***UPload*** (p239) for an easy way).

Options

-t Do only a test run. Files are checked out and changed, but rather than checking the changed files in again, the differences created are recorded.

Examples

```
UPmove -h
Usage:   UPmove [-t] [-c] [-r] [-u] [-d] [-n] <typ>
  Move files, and change references to them in all aips++
  code files. The script uses the UPfind and UPchange
  scripts (see there for details).
  <typ>.done and <typ>.error will contain the files successfully
  done or not.
Uses:    <typ>.def, <typ>.sel <typ>.cfnm, <typ>.todo
         <typ>_MVF.found, <typ>_MVF.spat, <typ>_MVF.rmpat
         <typ>_MV.sed, <typ>_MV.rcs, <typ>_MV.found
Creates: <typ>.done, <typ>.error, <typ>.diff
  -t will run a test mode
```

The next example is the start of a move of some files. The script is first run to get the information (and be able to edit it if necessary).

```
UPmove -nc spec
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/nfs/aips++/weekly]: /nfs/aips++/daily
```

```
Getting info...
Specify file pattern(s) rooted in AIPSROOT/code to find files
Example: trial/implement/Measures/M*.h
File pattern: trial/implement/Wnbt/Spectral*
File pattern: trial/implement/Wnbt/test/tSpec*
File pattern:
Specify egrep pattern for files to be removed from raw list found.
Example: [/]aipsview[/]
tmplinst files will always be removed
Remove glish files? (y|n) [y]:
Remove pattern:
Starting find...
16 files found in spec_MVF.raw
Starting selected removal...
16 files left in spec_MVF.found
16 files in spec.sel to be moved
Specify the sed patterns used to change the name of the moved files
Example: s#trial/#aips/#
Change pattern: s#/Wnbt#/SpectralComponents#
Change pattern:
Getting info...
Getting info...
Specify the message lines for the RCS log
Example: Moved includes from Quanta to Measures
Text line: Moved the Spectral fitting files from Wnbt to SpectralComponents
Text line:
Starting find...
16 files found in spec_MV.raw
Starting selected removal...
16 files left in spec_MV.found
Getting info...
```

The *UPlist* program can be used to check the files that will be changed (note that you can give the base type name, but that manually editing has to be done on the `spec_MV.found`):

```
UPlist spec
Getting info...
16 files found in spec.found
Starting list...
45 lines in spec.list
```

We now have a dry run with the *-t* test option:

```
UPmove -t spec
Getting info...
Need to move files first to new directory (y|n) [n]? y
16 files in spec.sel to be moved
```

```
16 files in spec.todo to be changed
amv -src ./trial/implement/Wnbt/Spectral2Element.cc
./trial/implement/SpectralComponents/Spectral2Element.cc
amv -src ./trial/implement/Wnbt/Spectral2Estimate.cc
./trial/implement/SpectralComponents/Spectral2Estimate.cc
...
Getting info...
16 files will be changed and checked in/out
Note: in test mode -- no actual file check in after change
./trial/implement/Images/DOimage2.cc started
/aips++/weekly/master/trial/implement/Images/DOimage2.cc,v  -->  DOimage2.cc
revision 15.18
done
...
16 files in spec_MV.done
0 files in spec_MV.error
```

Checking everything looks ok. We can now transfer all the files to aoc:

```
UPload spec
```

and do the actual move (while logged in to AIPS++ master):

```
UPmove spec
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/aips++/weekly]: /aips++/daily
Need to move files first to new directory (y|n) [n]? y
16 files in spec.sel to be moved
16 files in spec.todo to be changed

The rename command for ax_master is as follows:

15 304 src mv ./trial/implement/Wnbt/Spectral2Element.cc
./trial/implement/SpectralComponents/Spectral2Element.cc

Continue (y/n) [n]:
 /aips++/weekly/master/install/codemgmt/ax_master,v  --> ax_master
revision 15.525 (locked)
done
/aips++/weekly/master/install/codemgmt/ax_master,v  <--  ax_master
new revision: 15.526; previous revision: 15.525
done
Renamed master RCS version file
   from /aips++/weekly/master/./trial/implement/Wnbt/Spectral2Element.cc,v
     to
/aips++/weekly/master/./trial/implement/SpectralComponents/Spectral2Element.cc,v
```

```
...
./trial/implement/SpectralComponents/SpectralList2.cc started
/aips++/weekly/master/trial/implement/SpectralComponents/SpectralList2.cc,v
  --> SpectralList2.cc
revision 15.1 (locked)
done
/aips++/weekly/master/trial/implement/SpectralComponents/SpectralList2.cc,v
 <-- SpectralList2.cc
new revision: 15.2; previous revision: 15.1
done
16 files in spec_MV.done
0 files in spec_MV.error
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.8 *UPload*

Upload the data for the specified type name to the AIPS++ master computer.

Synopsis

UPload [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPload*** script uploads all relevant information from the given type name identifier to the aoc AIPS++ master (or another machine that can access the master tree with `amv`).
After transferring the information, login to the master, and run the relevant script(s) (like ***UPlock***, ***UPmove***).

Options

Examples

```
UPload -h
Usage:   UPload [-c] [-r] [-u] [-d] [-n] <typ>
  Upload the information specified by <typ> to the aips++
  master site. This enables to run relevant scripts that
  were tested locally there.
Uses:    <typ>.*
Creates: <typ>.scp
```

Using the information from the ***UPmove*** example, we get:

```
UPload spec
Getting info...
19 files in spec.scp
Uploading 19 files to wbrouw@aips2.nrao.edu:aips++/code/install/codemgmt ....
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.9 *UPtmpl*

Get a list of all files in all *tmplinst* directories.

Synopsis

UPtmpl [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The *UPtmpl* script lists all the files in all the `tmplst` directories in the AIPS++ code tree. The result is available in the *<type>.tmpinst* file.

Options

Examples

```
UPtmpl -h
Usage:   UPtmpl [-c] [-r] [-u] [-d] [-n] <typ>
  Get all files in all the tmplinst directories.
  The file list will be in <typ>.tmpinst
Uses:    <typ>.def
Creates: <typ>.tmpinst
```

A real run gives:

```
UPtmpl xc
Getting info...
Make sure you have a proper mktree built before proceeding...
Specify code tree root to be used [/nfs/aips++/weekly]:
Getting templated file list...
1729 files found in xc.tmpinst
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.10 *UPdup*

Do a duplicates template test on templates files in the AIPS++ code tree.

Synopsis

UPdup [-*s*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPdup*** script executes the ***duplicates*** program on templates files in the AIPS++ system. It checks the repositories in the casa packages against each other; checks the repositories and test templates for the casa packages, and it checks all other packages against the standard ***casa*** packages, creating a list with duplicates in the ***<type>.dup*** file. This output can be used to check if templates can e.g. safely be moved from some local test or application templates file to one of the standard ***casa*** packages. An ***-s*** switch is available to run in *system* mode, which will only indicate duplicates that in general could produce compilation and/or linking problems.

The package handling is done with the aid of in-built lists of 'standard' packages (***casapack***), packages that should be skipped (***skippack***) and packages that should each be checked against the standard ***casa*** packages (***checkpack***). The defaults can be overwritten by specifying one or more of these 3 variables in the ***<type>.pak*** file. After each run the used variables are saved in the ***<type>.pak*** file. A template can easily be derived by running with the ***-n*** switch.

Options

-s Execute in *system* mode. I.e. only show duplicates that are a real problem.

<type>.pak If the ***<type>.pak*** file is present in the directory from which the ***UPdup*** script is run, the checking behaviour can be influenced. Each line in the file contains a series of package names. The name follwing a '//' field gives the name of the variable to be overwritten, the rest are package names. I.e. if ***<type>.pak*** contains a line:

```
// checkpack atnf trialdisplay
```

then the duplicates check for the `atnf` and `trialdisplay` packages only will be done against the casa base packages.

Examples

```
UPdup -h
Usage:   UPdup [-s] [-c] [-r] [-u] [-d] [-n] <typ>
  Determine the template duplicates between the package and
  application/test programs. The result will be in <typ>.dup.
Uses:    <typ>.def   <typ>.pak
Creates: <typ>.dup   <type.pak
```

```
  -s show only the system template files that can cause
     problems in the aips++ system

  UPdup -s -u
  Assuming <typ> as 'xc'.
  Getting info...
  Starting package display...
  ...
```

...

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.11 *UPreident*

Do a reident on templates files in the AIPS++ code tree.

Synopsis

UPreident [-*a*] [-*z*] [-*t*] [-*c*] [-*r*] [-*u*] [-*d*] [-*n*] <type>

Description

The ***UPreident*** script executes the ***reident*** program on templates files in the AIPS++ system. By default it runs on the `casa` and other base library packages. Switches are available to execute it for all packages, and also to re-initialize the numbering. A list of succesful and unsuccessful files handled will be available in ***<type>.done*** and ***<type>.error***.

Options

-*a* Execute for all packages, rather than only for the `casa` base ones.

-*z* Apply the `-z` switch, i.e. renumber all templates

-*t* Do only a test run. Files are checked out and changed, but rather than checking the changed files in again, the differences created are recorded in ***<type>.diff***.

Examples

```
UPreident -h
Usage:   UPreident [-a] [-t] [-z] [-c] [-r] [-u] [-d] [-n] <typ>
  Do a reident for all the templates files in the casa basic library
  packages (unless -a switch).
  <typ>.done and <typ>.error will give the templates files
  done and not done.
Uses:    <typ>.def, <typ>.tsel
Creates: <typ>.done, <typ>.error, <typ>/log, <typ>.diff
  -a do all templates files in all packages
  -z use the -z switch on the reident program to renumber
     all the templates files.
  -t do a dry run, giving the differences in <type>.diff

  UPreident -tu
  Specify type [xc]:
  Getting info...
  42 templates files found in xc.tsel
  42 files will be changed and checked in/out
  Note: in test mode -- no actual file check in after change
  ./calibration/implement/CalTables/test/templates started
  U templates
```

```
  ./calibration/implement/_ReposFiller/templates started
  ...
```

...

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.12 *UPsed*

Apply a ***sed*** pattern to a file. This script is used by the other, main, ***UP*** scripts. It may be used stand-alone however.

Synopsis

UPsed [-t <log>] sed <filename in> [<filename out>]

Description

The ***UPsed*** script applies a number of sed patterns given in the ***sed*** file to an input file, and returns the changed input in the output file. The output file may be the same as the input file (if no output file given, the output file is assumed to be the same as the input file).
If a ***log*** file name is given as the first argument (preceded by a ***-t*** switch), the differences between the input and output file are added to the ***log*** file.

Options

-t <log file> Add the differences between input and output file to an existing log file

Examples

```
UPsed
Usage: UPsed [-t <log>] <sed> <filename in> [<filename out>]
```

Change a cat into a dog:

```
cat - > ! c.tmp
I am a cat today
cat - >! c.sed
s#cat#dog#
UPsed c.sed c.tmp
cat c.tmp
I am a dog today
```

```
cat - > ! c.tmp
I am a cat today
cat - >! c.sed
s#cat#dog#
touch c.log
UPsed -t c.log c.sed c.tmp
cat c.log
1c1
< I am a cat today
```

```
---
> I am a dog today
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

11.8.13 *UPinfo*

Obtain general information for the ***UP*** scripts. This script is a support script for the other ***UP*** scripts.

Synopsis

source UPinfo

Description

The ***UPinfo*** *source* script is a general aid script for the other, executable ***UP*** scripts. It analyses the switches given on the script command lines (and translates them into switches), and makes sure that the general information necessary to run the other scripts is available.

Options

Examples

```
UPinfo
Usage:source UPinfo
source UPinfo
script: Undefined variable.
```

Author

Original: 2000/09/20 by Wim Brouw, ATNF

Chapter 12

System verification

Utilities for verifying the integrity of AIPS++ [1].

12.1 *assay*

Invoke an AIPS++ test program or glish script, verify its output, and delete temporary files.

Synopsis

assay `[test program and arguments]`

Description

assay invokes an AIPS++ test program or glish script. If the program name has the suffix `.g` then it is a test glish script which is executed using the *glish* command. Otherwise it is a normal test program.
If the test program or glish script has an associated `.exec` file then *assay* invokes it using the *exec* command. Otherwise if it has an associated `.run` file then *assay* invokes it by simply starting it in another process.
If there is a corresponding `.out` file, *assay* compares the output of the program or script with that.

In comparing the output *assay* ignores text demarked by two lines starting with `>>>` and `<<<`. This may be useful as in the following example, where the test program prints a quantity such as the execution time which is not expected to be reproducible.

```
The answer to the question of life the universe and everything is: 42
>>>
Execution time: 5.4 aeons
<<<
```

[1] Last change: Id: SysVerif.tex,v 19.0 2003/07/16 04:18:27 aips2adm Exp

If there is a `.in` file associated with the test program then *assay* redirects `stdin` from it. If there are `.in_*` files then *assay* takes care they exist (temporarily) in the working directory, so the test program can find them there.

At the end *assay* takes care that all temporary files and directories are deleted. When the test program or script creates a temporary file or directory then it should be called `name_tmp*` where `name` is the name of the test program or glish script (e.g. `tTable_tmp1.data`).

It is permissible for a `.exec` to invoke *assay* (presumably with arguments for the test executable) – the potential infinite recursion is trapped.
Note that a `.run` has the advantage that one can massage the output before *assay* gets it.

Options

None.

Notes

- It is assumed that the test executable exists in the current `PATH`. If an environment variable called `ASSAYBIN` exists then `.:$ASSAYBIN` is added to the start of `PATH`. The AIPS++ system `bintest` directory is always appended to `PATH`.

Diagnostics

Status return values
`0`: success
`1`: test execution failed
`2`: test output disagreement
`3`: untested (as returned from a `.exec`)
`130`: interrupt

In addition, *assay* prints one of four POSIX 1003.3 compliant status messages referring to the result of the test execution:

`PASS (execution succeeded):` <test executable and arguments>
`FAIL (execution failure):` <test executable and arguments>
`PASS (output verified):` <test executable and arguments>
`FAIL (output not verified):` <test executable and arguments>

Examples

assay is used by the test makefile (`makefile.tst`) which uses *assay* to invoke the test executable without arguments. Where test programs do need to be invoked with particular arguments a `.exec` or `.run` must be provided. The following example, `tIncrementalStMan.exec`, uses *assay* to invoke *tIncrementalStMan* multiple times with different arguments:

```
#!/bin/sh
#-----------------------------------------------------------------------------
```

```
# Script to invoke tIncrementalStMan and verify its output.
#=============================================================================
  exec assay tIncrementalStMan "1000 && \
     tIncrementalStMan 2000 && \
     tIncrementalStMan 3000 && \
     tIncrementalStMan 4000 && \
     tIncrementalStMan 5000 && \
     tIncrementalStMan 0"
```

In this example note that *assay* is *exec*'d to save generating another process, and also that `tIncrementalStMan.out` is expected to contain the output from *tIncrementalStMan* for all six runs.

See also

Section 12.2, diagnostic makefile rules.

Author

Original: 1995/11/01 by Mark Calabretta, ATNF.

12.2 Diagnostic makefile rules

Use of the AIPS++ makefiles for various diagnostic purposes.

Synopsis

```
makefile
makefile.{app,aps,chk,doc,ftn,imp,pkg,scr,tst}
```

Description

The AIPS++ diagnostic targets are listed below by category. These lists are not exhaustive, but do aim to cover everything of practical use. In particular, they omit targets which are intended for the internal use of the makefiles.

A target is labelled as "recursive" if it causes *gmake* (p154) to be invoked in all subdirectories. It is "general" if it applies to all makefiles; such targets are defined in `makedefs` (p155). A target is "specific" if defined in a specific makefile.

Some targets such as `help` have a general meaning, the specific behaviour of which differs for specific makefiles. These are referred to as "general/specific" and where appropriate the details of a target's behaviour are described for each of the generic makefiles, for the top-level makefile (`top`), and the installation makefile (`ins`).

System diagnostic targets:

There are two classes of diagnostic target. The first class is self-referential in that it deals with the AIPS++ system and contains rules to report and/or verify `makedefs` variable definitions, report variables defined by the specific makefiles, print help information, and especially for debugging the makefiles. The rules to report and/or verify `makedefs` variables are actually defined in `testdefs` (p258).

- `command` : (general)
  This could be described as a "do-it-yourself" rule (without dependencies). It invokes a command or sequence of commands specified by the `COMMAND` variable defined on the *gmake* command line. This makes most sense if the command sequence uses `makedefs` variables.

- `printenv` : (general)
  Print out the environment as seen by the commands within makefile rules. This is especially useful for diagnosing problems related to makefile variables not being exported, unresolved recursively defined variables, and variables which cannot be exported to the environment because they contain non-alphanumeric characters.

- `eval_vars` : (general)
  Print variables specified by `VARS` on the *gmake* command line in a form suitable for *eval*'ing into the environment in Bourne shell. This is used in particular by *ax_master* (p115), *depend* (p152) and *updatelib* (p183) for getting `makedefs` variable definitions when invoked in stand-alone mode (when invoked via a makefile rule the required variables are explicitly `export`ed to them). An example of its use would be

```
eval `gmake -f $AIPSARCH/makedefs VARS="AR ARFLAGS" eval_vars`
```

This would create environment variables `AR` and `ARFLAGS` and give them the values of the `makedefs` variables of the same name.

- `diagnostics` : (general)
  This target simply invokes the `versions` and `test_global` targets. It is used by the code distribution system on cumulative updates to produce a status report for consortium installations (see *sneeze* (p85)).

- `versions` : (general)
  Print the installed version of certain utilities required by AIPS++, the main one being *gmake* itself.

- `show_all` : (general)
  This target invokes the `show_global` and `show_local` targets to show the value of all `makedefs` variables and all variables defined in the specific makefile.

- `test_all` : (general)
  Similar to `show_all` except that it invokes `test_global` instead of `show_global`. See `testdefs` (p258).

- `show_global` : (general)
  This target effectively invokes the `show_sys`, `show_prg` and `show_aux` targets to show all `makedefs` variables.

- `test_global` : (general)
  This target effectively invokes the `test_sys`, `test_prg` and `test_aux` targets to show and verify all `makedefs` variables. See `testdefs` (p258).

- `show_sys` : (general)
  Report the value of all system variables defined in `makedefs`.

- `test_sys` : (general)
  Report and verify the value of all system variables defined in `makedefs`. See `testdefs` (p258).

- `show_prg` : (general)
  Report the value of all variables defined in `makedefs` which are of immediate interest to AIPS++ programmers. This is particularly useful for reporting the compiler options set by the site-specific `makedefs` in response to use of the `OPT` variable or alternate programmer compilation flags.

- `test_prg` : (general)
  Report and verify the value of all variables defined in `makedefs` of immediate interest to AIPS++ programmers. See `testdefs` (p258).

- `show_aux` : (general)
  Report the value of all auxilliary variables defined in `makedefs`. (Certain of these may have been redefined in the specific makefiles.)

- `test_aux` : (general)
  Report and verify the value of all auxilliary variables defined in `makedefs`. See `testdefs` (p258).

- `show_local` : (specific)
  Report the value of all variables defined within the specific makefile. Typically these contain target and dependency lists.

- `show_vars` : (general)
  Report the value of the `makedefs` variables specified by `VARS` on the *gmake* command line. For example

```
cd $HOME/aips++/code/install
gmake VARS="AIPSSRCS" show_vars
```

  This would show all AIPS++ system sources in the current directory.

- `help` : (general/specific)
  Print an itemized summary of all general and specific targets categorized as "programmer", "system", or "diagnostic". The general targets are reported by `makedefs`, and the specific targets are appended by the specific makefile.

  Source code testsuite targets:

  The second class of diagnostic targets contains rules to compile and execute the test programs and verify their output.

- `cleansys` : (general/specific, recursive)
  The `cleansys` rule for the `tst` makefile cleans up files produced by the `runtests` rule. Refer to the description in section §10.9, p171.

- `cleanfail` : (specific)
  Delete reports for failed tests, see `runtests` below.

  – `tst`: Delete reports for failed tests in `$(BINTESTD)`. The deletion of these files will cause the `runtests` rule to selectively redo those tests. The list of failed tests is determined from `$(BINTESTD)/runtests.report`.

- `%.report` : (specific, pattern rule)
  Compile and exercise a test program.

  – `tst`: Compile a test program and invoke *assay* (p249) to exercise it. If necessary for the compilation, template instantiation files will be created from a `templates` file and compiled into an object library (which is retained). The executable and `.report` file, which contains the output from *assay*, are deposited in `$(BINTESTD)`. The `.report` file for each test program is used as a dependency file so the test will not be done if the `.report` file exists and is newer than the source files and libraries used to generate the test executable. The `cleanfail` rule can be used to selectively delete reports for failed tests. A one-line summary of the test result is reported by the rule which also collates and maintains these in `$(BINTESTD)/runtests.report`. The test executable will be deleted when gmake terminates (unless an up-to-date executable already existed in `$(BINTESTD)` when the rule was invoked).

- `runtests` : (general/specific)
  The general rule does nothing. Specific makefiles behave as follows:
  - `app`: (none)
  - `aps`: (none)
  - `chk`: (none)
  - `doc`: (none)
  - `ftn`: (none)
  - `imp`: Recurse into every package subdirectory.
  - `ins`: (none)
  - `pkg`: Recurse into the `implement` subdirectory.
  - `scr`: (none)
  - `top`: Recurse into every package subdirectory then print a summary of results.
  - `tst`: Invoke the `%.report` pattern rule for every test program in the directory. The intermediate object library containing template instantiations (if any) is deleted when all tests have been done.

Notes

- The rules to report and/or verify `makedefs` variables are actually defined in `testdefs` (p258).

Examples

After modifying any `makedefs` variable definitions,

```
gmake test_all
```

reports and verifies their values.

See also

The GNU `Make` manual.
The GNU manual page for ***gmake***.
AIPS++ variable names (§1.2, p8). ***gmake*** (p154), GNU make.
`makedefs` (p155), AIPS++ makefile definitions.
`testdefs` (p258), rules for reporting and validating AIPS++ makefile definitions.

12.3 *runtests*

Run the AIPS++ test programs and mail the results to the *aips2-inhale* email exploder with a summary to *aips2-workers*.

Synopsis

runtests

Description

runtests runs the AIPS++ test programs and mails the results to the *aips2-inhale* exploder and also sends a summary to *aips2-workers*. It invokes the `runtests` target described in section §12.2, p252 which stores its results in the `$BINTESTD` directory (§1.2, p8).

Options

None.

Notes

- The reported number of test program failures refers to the `aips` package only.
- Do not confuse the *runtests* utility with the `runtests` diagnostic makefile target (see §12.2, p252).

Diagnostics

Status return values
`0`: success
`1`: initialization error

Examples

The command

```
runtests
```

would invoke the test programs and mail the results to the various exploders.

See also

The manual page for *ci*(1), the RCS checkin command.
AIPS++ variable names (§1.2, p8).
Diagnostic makefile rules §12.2, p252.

Author

Original: 1997/01/29 by Mark Calabretta, ATNF.

12.4 testdefs

Report and test `makedefs` variable definitions.

Synopsis

```
testdefs
```

Description

`testdefs` is a GNU makefile which defines rules for reporting the values of `makedefs` variables and optionally testing their validity. `testdefs` is an adjunct to `makedefs` (p155); it is not meant to be used independently.

Test categories are:

- Check for the existence of a file.
- Check for the existence of a directory.
- Check for the existence of all directories in a path.
- Check for the existence of (object) libraries in a directory. Recognizes the syntax of the *-l* option to *ld*.
- Check for the existence of an executable. Reports the pathname if the executable was specified without pathname and also recognizes shell builtins.
- Check the syntax of a library control variable.
- Check the syntax of an AIPS++ link list variable.
- Check that a number lies within a specified range.
- Syntax checks for some control variables; `AUXILIARY`, `BINTEST`, `CONSORTIUM`, `DOCEXTR`, `DOCSYS`, `TESTOPT`, `TIMER`.

Note that about 40% of `makedefs` variables can't be sensibly tested. Chief among these are the variables which define compiler options.

Notes

- `testdefs` is primarily intended for the private use of `makedefs` which invokes its rules in either print or print-and-test mode via the `DO_TEST` variable.
- `testdefs` needs to be kept in step with `makedefs` if variables are added to or removed from the latter.

Files

`$AIPSARCH/testdefs` ...rules for reporting and testing variable definitions.

Diagnostics

`testdefs` associates one of three error levels with invalid variable definitions:

`ADVISORY`: A correction may be needed depending on the availability of resources such as libraries or modes of compiler operation.

This error level is typically associated with missing third-party object libraries whose absence may cause restricted compilation or link failures. Roughly a quarter of all errors fall into this category.

`WARNING`: An incorrect definition was found which should be fixed otherwise some rules will fail.

About two thirds of all errors fall into this category.

`SERIOUS`: A fundamental problem was found which will cause important rules to fail.

This error level is given for an invalid `AIPSROOT` or `AIPSARCH`, or problems which will cause catastrophic compile or link failures such as missing compilers.

See also

The *GNU make* manual.
The GNU manual page for *gmake*.
AIPS++ variable names (§1.2, p8).
gmake (p154), GNU make.
`makedefs adjuncts` (p160), Per-sourcefile makedefs definitions.
`makefiles` (p162), GNU makefiles used to rebuild AIPS++.

Author

Original: 1996/03/15 by Mark Calabretta, ATNF

12.5 *testsuite*

Maintain the most recent cumulative update to have passed the suite of AIPS++ test programs.

Synopsis

testsuite [-*f*#]

Description

testsuite maintains a copy of the most recent cumulative update to have passed the suite of AIPS++ test programs. This is the "dinkum" update fetched by specifying the *-D* option to ***inhale*** (p78).

testsuite is a special-purpose utility used only by *aips2mgr* on the master host. It operates as follows:

- Since it generally takes longer to compile and verify an update than the interval between successive invokations of ***exhale*** (p73), ***testsuite*** begins by preserving a copy of the `VERSION` file and the current cumulative update file in `$MSTRFTPD`. It does this by creating hardlinks to each of these files within `$AIPSMSTR/testsuite/`, a special directory created for its exclusive use. (***exhale*** does not delete a cumulative update file with multiple links to it.)
- ***testsuite*** then does a cumulative ***inhale***, mailing the log to the *aips2-inhale* exploder.
- When the ***inhale*** has finished ***testsuite*** invokes ***gmake runtests***, preserving the log in `$BINTESTD/runtests.log`.
- It then mails the summary, `$BINTESTD/runtests.report`, to the *aips2-workers* exploder.
- If the test suite passed, ***testsuite*** installs the cumulative update as the new dinkum update by
  - Preserving the hardlink to the cumulative update file as `$AIPSMSTR/testsuite/dinkum`. This hardlink is never accessed by the code distribution system, its only purpose is to stop ***exhale*** from deleting the linked-to file in `$MSTRFTPD`.
  - Preserving the `VERSION` file as `$MSTRFTPD/DINKUM`. This file substitutes for the `VERSION` file when ***inhale*** is invoked with the *-D* option.
- Finally ***testsuite*** prints `$BINTESTD/runtests.report` and `$BINTESTD/runtests.log` on `stdout`.

aips2mgr in Socorro invokes ***testsuite*** once a week via a ***cron*** job in place of ***inhale*** *-c*. It redirects the output to `$AIPSROOT/testsuite.log` and also mails it to the *aips2-inhale* exploder.

Options

-f# Allow the specified number of failures when deciding whether the test suite passed or not.

Notes

- ***testsuite*** is logically a function of *aips2adm*, the AIPS++ master administrator. However, *aips2adm* cannot exercise the test programs since it does not maintain a functioning AIPS++ system. Thus the responsibility devolves to *aips2mgr* on the master host.

  This puts *aips2mgr* on the master host in a special position. ***testsuite*** is unique in the AIPS++ system as being the only utility designed for use by a particular *aips2mgr*.

- The ***testsuite*** executable resides in the `bin` subdirectory of *aips2mgr*'s home directory in Socorro.

- "Dinkum" as in "fair dinkum" is an Australian colloquialism meaning true, honest or genuine.

Diagnostics

Status return values
  `0`: success
  `1`: initialization error

See also

AIPS++ variable names (§1.2, p8).
Section §1.4, p9, AIPS++ accounts and groups.
Section §12.2, p252, diagnostic makefile rules.
assay (p249), invoke an AIPS++ test program and verify its output.
exhale (p73), AIPS++ code export utility.
inhale (p78), AIPS++ code import utility.
runtests (p256), run the AIPS++ test programs.
sneeze (p85), AIPS++ system rebuild utility.

Author

Original: 1996/09/18 by Mark Calabretta, ATNF.

Chapter 13

Central services

This chapter [1] describes centrally provided AIPS++ services. These are associated with the *aips2adm* account on the master host, *aips2.nrao.edu* in Socorro. The relevant utilities are installed in `$(MSTRETCD)` on the master.

13.1 AIPS++ web services

This section describes web services provided by the AIPS++ master host.

Master services

The AIPS++ master host, *aips2.nrao.edu*, is set up as the web server for AIPS++ at NRAO Socorro. The master web server directly provides a number of services which may only be linked to by other AIPS++ web servers:

- The master RCS repository.
- The archive RCS repository.
- The AIPS++ anonymous *ftp* areas.
- The AIPS++ email exploder archives.
- Separately released AIPS++ software components:
  - *cxx2html*.
  - FITS classes.
  - Glish.

The AIPS++ master home page is at http://aips2.nrao.edu.

[1] Last change: Id: CentServ.tex,v 19.0 2003/07/16 04:18:29 aips2adm Exp

Server configuration

aips2.nrao.edu runs NCSA's ***httpd***. The essential features of the server configuration are as follows:

- The server is installed in `/home/tarzan/httpd/...`.
- The ***httpd*** `DocumentRoot` directory is `/home/tarzan`.
- The ***httpd*** `ScriptAlias` directory is `/home/tarzan/httpd/cgi-bin`.
- `/home/tarzan/robots.txt` contains entries which exclude web crawlers from certain parts of the document hierarchy, notably the email archives.

The AIPS++ links are enabled via unix symlinks in `DocumentRoot`:

```
aips++/archive  -> /export/aips++/archive/
aips++/ftp      -> /export/aips++/pub/
aips++/mail     -> /export/aips++/Mail/
aips++/master   -> /export/aips++/master/
aips++/RELEASED -> ftp/RELEASED/
```

plus there are also links to the local AIPS++ installations:

```
aips++/daily  -> /aips++/daily/
aips++/weekly -> /aips++/weekly/

aips++/code   -> daily/code/
aips++/docs   -> daily/docs/
```

The AIPS++ links are presented in a HTML document `/home/tarzan/index.html`.

See also

Section §13.2, p265, AIPS++ email exploders.
http://hoohoo.ncsa.uiuc.edu, NCSA HTTPd, etc.
http://info.webcrawler.com/mak/projects/robots/robots.html, Web robots home page.

13.2 AIPS++ email exploders

This section describes how the AIPS++ electronic mail exploders are set up.

Exploder lists

The sources of the AIPS++ exploder lists reside in `$(AIPSCODE)/admin/system` in files with a `.list` suffix, for example `aips2-workers.list`. AIPS++ programmers may modify these lists using ***ao*** (p97) and ***ai*** (p93) to check them in and out like any other source file. The lists are installed in `$(MSTRETCD)` by the `$(MSTRETCD)` makefile which is invoked by ***exhale*** (p73) several times a day.

AIPS++ exploder email is addressed to *nrao.edu*, for example *aips2-workers@nrao.edu*. NRAO's email system is set up so that this mail is received by *aips2.nrao.edu*, the AIPS++ master host, which distributes email to the list by virtue of entries of the following form in `/etc/aliases`

```
aips2-workers: :include:/export/aips++/master/etc/aips2-workers.list
```

Exploder archives

The first entry in every AIPS++ exploder list is an email alias which is used to archive a copy of the mail. These aliases have names of the form `*-log`, where the asterisk corresponds to the exploder name. For example, `aips2-workers.list` contains `aips2-workers-log`.

The `-log` aliases are defined in `/etc/aliases` on *aips2.nrao.edu* so that the received mail is piped to a utility, ***aipsmail*** (p267), which does the archiving using the publically available MH (message handler) system. For example, the entry for the *aips2-workers* exploder is

```
aips2-workers-log: "| /export/aips++/master/etc/aipsmail aips2-workers"
```

The MH folders reside in directories of the same name as the exploder in the `Mail` directory beneath *aips2adm*'s home directory. For example the *aips2-workers* archive resides in

```
/export/aips++/Mail/aips2-workers
```

MH stores each email in a separate file in this directory and gives them numerical names corresponding to the sequence number of the message.

Mail which is more than 14 days old is deleted from the *aips2-inhale* exploder every evening by an *aips2adm* ***cron*** job running on *aips2.nrao.edu*.

Web interface to the exploder archives

The MH mail folders in which the exploder archives reside are made publically accessible on the web via a symlink in the ***httpd*** `DocumentRoot` directory (`/home/tarzan`, see section §13.1, p263):

```
/home/tarzan/aips++/mail -> /export/aips++/Mail
```

The AIPS++ home page contains a link to an HTML document, `$(AIPSDOCS)/html/email.html`, which in turn contains links to email folder indexes generated by ***scanhtml*** (p278). ***scanhtml*** invokes the MH ***scan*** command to produce an index and then converts it to HTML. ***scanhtml*** is invoked by ***aipsmail*** whenever it receives mail so the indexes are always current.

Apart from containing links to each item of exploder mail, the indexes also contain links to the currently installed copies of the exploder lists, and a link to a form which allows searching of the exploder archive.

The link to the exploder list is implemented via a unix symbolic link called `list` within the mail folder directory. For example

```
/export/aips++/Mail/aips2-workers/list -> ../../master/etc/aips2-workers.list
```

Mail searching is implemented via a CGI (Common Gateway Interface) script, ***pickhtml*** (p275), which generates an HTML form to obtain options for another CGI script, ***scanpick*** (p279). ***scanpick*** invokes the MH ***pick*** command to search the archive folder and converts the ***pick*** output to an HTML index. The CGI scripts are web-enabled via symlinks in the ***httpd*** `ScriptAlias` directory (`/home/tarzan/httpd/cgi-bin`):

```
/home/tarzan/httpd/cgi-bin/pickhtml -> /export/aips++/master/etc/pickhtml
/home/tarzan/httpd/cgi-bin/scanpick -> /export/aips++/master/etc/scanpick
```

Refer to the AIPS++ email exploder web page for a list of currently active exploders.

See also

The unix manual page for ***cron***(1).
The manual page for ***mh***(1), the message handler system.
The manual page for ***pick***(1), the MH search command.
The manual page for ***scan***(1), the MH index command.
AIPS++ code management configuration (§1.1.4, p5).
AIPS++ variable names (§1.2, p8).
Section §1.4, p9, AIPS++ accounts and groups.
ai (p93) AIPS++ code checkin utility.
ao (p97) AIPS++ code checkout utility.
exhale (p73), AIPS++ code export utility.
Section §13.1, p263, AIPS++ master web services.
aipsmail (p267), AIPS++ email exploder archive utility.
pickhtml (p275), AIPS++ HTML form generator for ***scanpick***.
scanhtml (p278), AIPS++ HTML indexing utility for exploder email.
scanpick (p279), AIPS++ HTML interface to ***pick***.
http://hoohoo.ncsa.uiuc.edu/cgi, Common Gateway Interface.

13.3 *aipsmail*

Archive AIPS++ exploder mail.

Synopsis

`| aipsmail [folder]`

Description

aipsmail archives AIPS++ electronic mail. It is invoked directly by ***sendmail*** on *aips2.nrao.edu* via entries in the `/etc/aliases` file as described in §13.2, p265.

aipsmail expects to receive the text of a mail message on `stdin`. It first invokes ***rcvstore*** to store this message in the specified MH mail folder – `general` by default. It then invokes ***scanhtml*** (p278) to update the HTML index for the folder.

Options

None.

Notes

- Since ***sendmail*** normally invokes commands using the identity of the person who sent the mail, ***aipsmail*** needs to run as setuid to *aips2adm*.

Diagnostics

Status return values
`0`: success

See also

The manual page for ***mh***(1), the message handler system.
The manual page for ***rcvstore***(1), the MH command to store mail.
Section §1.4, p9, AIPS++ accounts and groups.
Section §13.2, p265, AIPS++ email exploders.

Author

Original: 1995/07/18 by Mark Calabretta, ATNF

13.4 *asco*

Report AIPS++ sources currently checked out.

Synopsis

asco

Description

asco reports AIPS++ sources currently checked out. A general summary is posted to the *aips2-workers* exploder, and personal summaries are sent to each user who has a file locked.

asco is invoked as an *aips2adm* *cron* job on *aips2.nrao.edu* each Sunday evening.

Options

None.

Notes

- *asco* uses `alog -m -L -t`.

Diagnostics

Status return values
`0`: success
`1`: initialization error

See also

The unix manual page for *cron*(1).
Section §1.4, p9, AIPS++ accounts and groups.
ai (p93), AIPS++ code checkin utility.
ao (p97), AIPS++ code checkout utility.
alog (p100), AIPS++ change log reporting utility.

Author

Original: 1995/03/20 by Mark Calabretta, ATNF

13.5 *astat*

Collate AIPS++ revision statistics from an ***alog*** report.

Synopsis

alog [options] | ***astat***

Description

astat is a ***sed/awk*** filter which collates revision statistics for each AIPS++ programmer from an ***alog*** (p100) report. It reports separately the number of new files checked in, revisions of previously checked in files, and the total of these.

For each revision made (but not for the initial checkin) RCS provides a rough indication of the extent of the change by reporting the number of lines added and the number of lines deleted. Cumulative totals for these are provided in the output, but note that these numbers can be misleading. Large values may, for example, result from checking in a changed PostScript file, or test program output.

Options

None.

Notes

- The AIPS++ master and slave RCS files contain only the current and preceeding versions (see ***exhale*** (p73)). The archive on the AIPS++ master host, *aips2.nrao.edu*, contains all versions up to the base release of the current major version.
- Account names for some checkins from remote sites are normalized to standard form.
- Only a fraction of the efforts of AIPS++ workers is reflected in the checkin statistics. This fraction may be quite low for some but high for others.

Examples

```
alog -b --package=aips | astat
```

Collects statistics for the `aips` package for the latest AIPS++ major version.

See also

alog (p100), AIPS++ change log reporting utility.

Author

Original: 1994/02/10 by Mark Calabretta, ATNF

13.6 *help2tex*

Generate standard LaTeX or *atoms.g files from a .help file.

Synopsis

help2tex [options] | *help2tex*

Description

help2tex is a perl script which translates aips2help.sty commands and enviornments into standard LaTeX and into glish help atoms. *help2tex* also does some formatting/rearranging of the help text to provide a bit of consistency for the user's manual.

Options

-noheaders Do not put standard latex2e document headers around the help text.

-glish -package packagename Generate glish help atoms (glish attributes) which belong to package packagename.

Notes

- Output goes to standard out so it need to be redirected to be saved.
- Most folks should run thier .help files through *help2tex* to check the output before checking the .help file into the system.

Examples

```
help2tex measures.help > measures.htex
help2tex -noheaders aips.help > aips.htex
help2tex -glish -package aips > aipsatoms.g
```

- Turns aips2help.sty functions and enviornments into standard LaTeX, can be run through latex2e.
- Generate standard LaTeX, without the headers (used in the make system)
- Create the atoms.g file used in generating command-line glish help.

See also

- Note 197, How to Write an AIPS++ Application, Documentation Your Application
- Note 215, AIPS++ Documentation

Author

Original: 1997/11/12 by Wes Young, NRAO

13.7 *parseform*

Decode HTML forms output.

Synopsis

eval `parseform`

Description

parseform decodes the output from an HTML form reporting it on `stdout` in a format suitable for *eval*'ing into the environment in a Bourne shell CGI script. Both the `GET` and `POST` methods of HTML form output are supported.

Options

None.

Notes

- *parseform* only works when called from a CGI script. Note that the form output generated by the `POST` method appears on `stdin` and *parseform* consumes this. Form output generated by the `GET` method is obtained from environment variable `QUERY_STRING`.
- *parseform* is implemented as a *perl* script.

Diagnostics

Status return values
  0: success

Examples

scanpick uses the following to get and parse the form output generated by *pickhtml*:

```
# Get the form output.
  eval `$MSTRETCD/parseform`
```

See also

Section §13.2, p265, AIPS++ email exploders.
pickhtml (p275), AIPS++ HTML form generator for *scanpick*.
scanpick (p279), AIPS++ HTML interface to *pick*.
http://hoohoo.ncsa.uiuc.edu/cgi/forms.html, Decoding FORMs with CGI.

Author

Original: 1996/05/21 by Mark Calabretta, ATNF

13.8 *pickhtml*

HTML forms based interface to search an AIPS++ email archive.

Synopsis

pickhtml `[folder]`

Description

pickhtml is a CGI script which generates an HTML form to obtain options for the MH ***pick*** command to search an AIPS++ email archive folder. These options are passed on to ***scanpick*** (p279) via the `POST` method.

scanhtml (p278) creates a link to ***pickhtml*** in the HTML index to each AIPS++ email archive folder.

If not supplied the folder name defaults to `general`.

Options

None.

Notes

- ***pickhtml*** is web-enabled via a symlink in `/home/tarzan/httpd/cgi-bin`, the ***httpd*** `ScriptAlias` directory:

```
/home/tarzan/httpd/cgi-bin/pickhtml -> /export/aips++/master/etc/pickhtml
```

Diagnostics

Status return values
  `0`: success

See also

The manual page for ***mh***(1), the message handler system.
The manual page for ***pick***(1), the MH search command.
Section §13.2, p265, AIPS++ email exploders.
scanpick (p279), AIPS++ HTML interface to ***pick***.

Author

Original: 1996/05/21 by Mark Calabretta, ATNF

13.9 *reap*

Collate and disseminate AIPS++ weekly reports.

Synopsis

reap

Description

reap collates and disseminates AIPS++ reports. It maintains a timestamp file, `.reap.time`, within the *aips2-reports* mail folder, collects all reports newer than the timestamp, strips out the mail headers, concatenates them and posts the result to the *aips2-weekly-reports* exploder. It is invoked regularly by an *aips2adm* *cron* job running on *aips2.nrao.edu*.

Options

None.

Diagnostics

Status return values
- `0`: success
- `1`: initialization error

See also

The unix manual page for *cron*(1).
Section §1.4, p9, AIPS++ accounts and groups.
Section §13.2, p265, AIPS++ email exploders.
`report_form`, AIPS++ weekly report form.

Author

Original: 1996/05/05 by Mark Calabretta, ATNF

13.10 report_form

AIPS++ weekly report form.

Synopsis

/usr/lib/sendmail -t < /export/aips++/master/etc/report_form

Description

report_form is sent to the *aips2-weekly-reports* email exploder each week by an *aips2adm* ***cron*** job running on *aips2.nrao.edu*. It solicits a report which is to be sent to the *aips2-reports* exploder where they are collected by ***reap*** (p276) which concatenates and mails them back to *aips2-weekly-reports*.

Notes

- The report_form is fed directly to ***sendmail***.

See also

The unix manual page for ***cron***(1).
Section §1.4, p9, AIPS++ accounts and groups.
Section §13.2, p265, AIPS++ email exploders.
reap (p276), AIPS++ report collator.

Author

Original: Jim Horstkotte, NRAO

13.11 *scanhtml*

Produce an HTML index of an AIPS++ mail folder.

Synopsis

scanhtml [folder]

Description

scanhtml produces an MH *scan* listing of an AIPS++ exploder mail archive folder and converts it to an HTML index with links to the individual messages. The index, `index.html`, is deposited in the folder directory. Any pre-existing index is overwritten.

As described in §13.2, p265 the index generated by *scanhtml* also contains links to the currently installed copies of the exploder lists, and a link to a form which allows searching of the exploder archive.

If not supplied the folder name defaults to `general`.

Options

None.

Notes

- *scanhtml* is invoked by *aipsmail* (p267) immediately after new exploder mail is archived. It is also used by the *aips2adm* *cron* job which deletes old mail from the *aips2-inhale* archive. It can be invoked manually (by *aips2adm*) at any time if required.

Diagnostics

Status return values
`0`: success

See also

The unix manual page for *cron*(1).
The manual page for *mh*(1), the message handler system.
The manual page for *scan*(1), the MH index command.
Section §1.4, p9, AIPS++ accounts and groups.
Section §13.2, p265, AIPS++ email exploders.

Author

Original: 1995/07/18 by Mark Calabretta, ATNF

13.12 *scanpick*

Produce an HTML index of selected mail in an AIPS++ folder.

Synopsis

scanpick [folder]

Description

scanpick is a CGI script which produces a *scan* listing of messages in an AIPS++ MH email archive folder selected according to *pick* arguments.

The *pick* arguments are acquired by another CGI script, *pickhtml* (p275), which generates an HTML form and passes the result to *scanpick* via the POST method.

scanpick uses an auxiliary *perl* script called *parseform* (p273) to decode the form output.

Options

None.

Notes

- *scanpick* is web-enabled via a symlink in /home/tarzan/httpd/cgi-bin, the *httpd* ScriptAlias directory:

  ```
  /home/tarzan/httpd/cgi-bin/scanpick -> /export/aips++/master/etc/scanpick
  ```

Diagnostics

Status return values
0: success

See also

The manual page for *mh*(1), the message handler system.
The manual page for *pick*(1), the MH search command.
The manual page for *scan*(1), the MH index command.
Section §13.2, p265, AIPS++ email exploders.
pickhtml (p275), AIPS++ HTML form generator for *scanpick*.
parseform (p273), AIPS++ decoder for HTML forms output.

Author

Original: 1996/05/21 by Mark Calabretta, ATNF

Index