ASTROPHYSICS and SPACE RESEARCH GROUP
DEPARTMENT OF PHYSICS AND SPACE RESEARCH
UNIVERSITY OF BIRMINGHAM
SYS/001
Asterix System Note
ASTERIX System Note 001
David J Allan
27 May 1996

ASTERIX System Architecture

Contents

1 Introduction

This document explains the system architecture of the ASTERIX package and in particular the way in which make files are used. It also highlights some of the reasons why things are done the way they are. Sometimes things me seem more complicated than they need be, but before changing anything fundamental be sure to test that it will work on all the machines supported (be especially wary of vendor specific make file features), and that it will work in both the development and publicly released systems.

1.1 The Basics

All Starlink software uses the `SYSTEM` environment variable to control machine dependent aspects of software construction. It is normal for accounts doing software development to set this up in their `.cshrc` file. The values *currently* in use are,

sun4_Solaris SUN SPARCs running Solaris

alpha_OSF1 DEC Alphas running OSF1

ix86_Linux Generic PC architecture running Linux

In addition we have supported **sun4** (SUNs running SunOS) and **mips** (DECstations running Ultrix) in the past, although we don't any longer.

All Starlink software uses the UNIX system utility `make` to build its software. The associated make files are written in a generic fashion with the aim of containing no system dependent statements. Operations which are potentially non-portable (eg. the command to replace an object module in an archive) are tokenised. Each Starlink package then has a Bourne shell script called `mk` which sets up the values of these tokens depending on the value of `$SYSTEM`. All make files are therefore invoked using this `mk` script rather than running `make` directly.

Make files define a number of targets. The ones used by Starlink are usually high level operations such as 'build the system from the source code', or 'install this package into the Starlink tree'. These high level targets are called *primary targets*. The *secondary* targets are the actual files which are built or modified by the system. Generally it is good practise not to access the secondary targets directly. A `make` target is activated if the file which the target specifies is to be built is out of date with respect to its dependencies. A target with no dependencies is always activated.

For primary targets it is not unusual to have dummy or zero length files which due to their date stamp in the file system can be used as date carrying flags. Starlink software usually maintains 3 such files in each directory called `.BUILT`, `.INSTALLED` and `.TESTED`. These flag files tell the system in which of 4 potential states the package exists. In ASTERIX, because we build our volatile files from one copy of the source code, these files are suffixed with the value of `$SYSTEM`. Starlink aren't too keen on this, but who cares, and they get flack for having multiple copies of software.

1.2 Revision Control

The ASTERIX development system uses **sccs**, the UNIX *s*ource *c*ode *c*ontrol *s*ystem. This is done for two very different reasons,

- Almost unbelievably the text archive format is not portable between SunOS and Solaris, and not up-datable between Solaris and OSF. A container is required for text modules, and updates to tar archives are not universally supported.
- We need some way of preserving old versions of code which accessible with resorting to system backups.

The result is that each library or application group subdirectory in ASTERIX contains an SCCS archive. Some useful SCCS commands are,

Command	Purpose
mkdir SCCS	Start a new SCCS archive
sccs create <module>	Create a new module
sccs get <module>	Extract a read only copy of a module
sccs edit <module>	Extract a writable copy of a module
sccs prs <module>	Display revision history of a module
sccs get -r<rev> <module>	Extract a particular revision
sccs -r<rev> SCCS	Increment version counter in archive

Typical mistakes using SCCS include,

- Making a read only copy of a file, editing it (because jed allows this), and replaceing it. Effect is to change object module correctly but *does not* update the source archive. Result is that next time you rebuild the library the change goes away!
- Deleting the writable copy of a module. Simply re-extract the module using the sccs get -r mode (a simple sccs get fails because SCCS believes the module is write locked), change the protection using chmod w+ and then replace it.

If you make a change which subsequently turns out to have been ill-advised then you can recover old versions by discarding subsequent revisions. See the manual for details. You can also recover the whole archive to a particular date.

It is important to realise that only the development system uses SCCS – Starlink just get tar files containing the source modules and build it from those. This duality is controlled by the AST_SYS_DEV environment variable which causes the mk script to make alternate token substitutions. Starlink build ASTERIX without setting this variable, we define a value for it (which is itself irrelevant) in the sysdev procedure.

1.3 The Primary Targets

Lets go back to the state files BUILT_$SYSTEM etc. A Starlink package (and for us, any ASTERIX directory) can exist in 4 states, unbuilt (or source), built, installed and tested. The system is in the first state if you unpack the source code distribution. The primary targets can now be understood in terms of transitions between these states.

build Changes the state from unbuilt to built by constructing all the volatile, usually system dependent, files (eg executables) Success is marked by the presence of the `.BUILT_$SYSTEM` file.

unbuild Goes from built back to unbuilt by deleting the volatile files and the built flag file.

check Check that all the files that will be needed for a 'build' are in fact present. Generally a test that the 'tar' file was not corrupted

clean Deletes any temporary files created by the 'build' target. Generally a null task in ASTERIX

install Change the state from built to installed. Involves moving and sometimes editing the volatile files into different directories. Create the `.INSTALLED_$SYSTEM` flag file. Generally the built volatiles are replaced with soft links to their installed counterparts to save disk space.

deinstall Recover the built system from the installed state. It is quite important that this pair of targets are well written such that if the install partially fails then the built system can be correctly recovered.

export Builds a `tar` file containing all the built files for the architecture specified by the `SYSTEM` environment variable. The tar file is created compressed in the top-level directory with name `asterix_$SYSTEM.tar.Z`.

export_source Builds a `tar` file containing all the source and make files required to generate the binary built system and the developer system. The file is created compressed in the top-level directory with name `asterix.tar.Z`.

test Run a test procedure and create the test flag file. Not used by ASTERIX.

In addition ASTERIX make files support a few more primary targets,

help List the primary targets available to this directory

developer Create the development system from the normal source distribution. This involves setting up the SCCS directories.

makes Recreate the `Makefile` for each of the dependent sub-directories of the current directory. Invoked at the top level of the ASTERIX distribution this will rebuild all the make files except the top level one (which is a special case anyhow).

showtree Display the directory structure from the current directory downwards. Invoked from the top-level shows the ASTERIX directory tree

rebuild Just performs a **unbuild**, then a **build** and finally a **clean**.

get_s_files Lists source files in the directory and all sub-directories. This target is used by the `build_export` target to construct the releasable files.

get_b_files Lists built files in the directory and all sub-directories. Used for building the binary release.

2 The ASTERIX File Structure

The ASTERIX top level development directory contains a number of *modules* each of which has its own directory, plus one directory for each architecture on which the system is built. The latter directories have the same name as their corresponding `$SYSTEM` value. Source files for modules are broken up into different directories depending on their type. The system has been designed so that a directory contains only one class of file (eg. applications, script or data) which enables the make-files required to build those files be parameterised using the GEN_make facility.

2.1 The `build` system

Each module sub-directory constructs built files from the source files (even if they are only copies), and moves them into the appropriate sub-directory of the system dependent directory for that architecture. There are only 5 such sub-directories.

dates Text date stamp files. These are using to store details of build times and macro definitions extant at the time of build. One for every directory in ASTERIX.

bin Executables and scripts. This is the directory which is added to the user's `PATH` string.

etc Everything else needed by the run-time system, apart from help text and documentation. Contains data files, non-executable scripts (eg. Tcl routines) and anything else you care to mention.

etc/sys This contains a small subset of files needed by the ASTERIX system. They are those which control the modular aspect of the system, and include the module start up scripts and include file tables. Their presence in this directory greatly speeds up the ASTERIX startup because many fewer directory entries need to be scanned compared with keeping them in `etc`.

etc/docs Contains all ASTERIX internal documentation, apart from SUN 98, the guide to the whole package.

lib Object archives and object modules

help Online help info

The entire ASTERIX system runs from within these directories, and relies on nothing outside. This results in duplication of the contents of the `etc` directory which is unnecessary as the files are portable, but greatly simplifies the overall architecture of the system, and satisfies Starlink requirements for architecture separation.

2.2 The `install` system

The ASTERIX development system runs in the built state, but for external consumption Starlink insist we distribute our files into the Starlink directory structure. This is done by moving the built files from the build system directory `xxx` to `/star/xxx/asterix`. Most files can simply be copied, but some scripts have to be edited. When the **install** target moves the files is replaces the copies in the build directories with soft links to the installed files. The **deinstall** target destroys these links and moves the files back to the build directories.

2.3 Modules

The module is the smallest optional unit in ASTERIX, ie. it is the unit in which sub-packages of ASTERIX are distributed. All modules reside at the top level of the file structure, although not all top level directories need be modules.

2.4 The Directory Classes

The various classes of ASTERIX directories are discussed below. There is nothing to stop new classes being defined, but you should ensure that they behave according to the general principles.

2.4.1 The General Principles

All ASTERIX directories can contain sub-directories. The list of such directories is defined by the token **SUBDIRS**. The order specified in this token is the order in which the sub-directories are built and installed – these operations take place *before* the build and install phases of the directory which is the container. The **SUBDIRS** token defines another token called **REV_SUBDIRS**, which is the order in which sub-directories are unbuilt and deinstalled.

A useful target supported by *all* ASTERIX directory classes is the `showtree` target. This recursively displays the names of all the directories in the **SUBDIRS** list, and all their sub-directories, and so on until a directory is reached which has no sub-directories. The output looks something like,

```
% cd $AST_ROOT
% mk showtree

  dev/
    etc
    scripts
  kernel/
    hed/
      scripts
    lib/
      etc
      inc
      scripts
    util/
      scripts
  ....
  rosat/
    scripts
```

2.4.2 The `etc` Class

The `etc` class is the simplest directory class. Source files are simply copied to the `etc` subdirectory of `$AST_ROOT/$SYSTEM` in the build phase, and `$INSTALL/etc/asterix` in the install phase.

Often the files which are specified by the `etc` directory are calibration files, which are perhaps computed by some application or supplied by an external agency. Each such file should have a subdirectory containing either the code required to make the file or relevant documentation. This directory is not part of the system as far as building and installing is concerned, and is ignored by the `mk` system. An example of this is the ASCA SIS psf calibration, which lives in `kernel/etc`. A subdirectory called `SISpsfGenerator` exists which contains the code which makes this file.

The only mandatory token required by the `etc` class make file is one called **ETC_FILES**. You can list all the files under this token or you can split them, which is neater. For example,

```
files BITMAPS marilyn.xbm pamella.xbm claudia.xbm

files COLTABS mary.sdf mungo.sdf midge.sdf

#
# The mandatory token
#
files ETC_FILES :BITMAPS: :COLTABS:
```

2.4.3 The `docs` Class

A simple variant of the `etc` class which simply copies its files into the directory `$AST_ROOT/$SYSTEM/etc/docs`. The only token used is **DOC_FILES** which is a list of the documents to be copied. Current practice in ASTERIX is to have all user documents available in an installation regardless of the modules actually installed. System maintainance and programmer documentation may however be module specific.

2.4.4 The `scripts` Class

Contains *public* and *private* scripts. Public scripts are executable files which are built to the directory `$AST_ROOT/$SYSTEM/bin`, and installed to `$INSTALL/bin/asterix`. Private scripts are auxiliary files ("subroutines"), which are not intended to be directly referenced by the user. These are built to `$AST_ROOT/$SYSTEM/etc` and installed to `$INSTALL/etc/asterix`.

There are 4 types of supported scripts – C shell scripts, Bourne shell scripts, ICL procedures and Tcl procedures. The first three require no special treatment by the `mk` system, but the Tcl files require the location of the `tclsh` interpreter and the ADAM Tk library edited in. These locations are specified by token substitutions in the `mk` script.

Only those tokens denoting source for the class of scripts in a particular directory are mandatory, eg.

```
files PRIVATE_TCL_SRC sub1.tcl doit.tcl

files PUBLIC_ICL_SRC  mickey.icl mouse.icl language.icl
```

We have no Bourne shell scripst so we don't have to specify the **PUBLIC_SH_SRC** or the **PRIVATE _SH_SRC** tokens.

2.4.5 The tools Class

The `tools` class supports C and Fortran stand-alone programs. These tools are used to support ASTERIX operations. Examples include the file streamer `at_strm` (used by GEN_make) and the help reader. The requirement on the linking of these tools is that they must *not* depend on any other part of ASTERIX. They may require linking against Starlink libraries. Executables are built to the `$AST_ROOT/$SYSTEM/bin` directory and installed in `$INSTALL/bin/asterix`.

2.4.6 The inc Class

The purpose of this class is to support C and Fortran include files. The C ones are no problem, it's the Fortran ones that cause the hassle. All ASTERIX Fortran uses the old VMS style logical names for include files. These are translated by the `v2u_include` script into absolute UNIX file names. This process is performed by the Starlink `forconv` program which reads in a file called `include.sub` which is a two column text file containing the mappings from VMS style include file names to UNIX ones. The file of these mappings is constructed on the fly by `v2u_include`, and destroyed after translation (although their is a persistent mode to support bulk compilation). The mapping file is made by concatenating all the mapping files generated for each of the ASTERIX `inc` directories. These mapping files are stored in `$AST_ROOT/$SYSTEM/etc/sys` in the built system.

It is worth mentioning that C modules in ASTERIX have to have their include file paths specified somehow. This is done in two ways. Include paths which are system dependent are included in the **mk** script using the `CFLAGS` environment variable. The `replace` script in `dev/scripts` then adds the current directory, the ASTERIX library include directory and the Starlink include directory to the include path list, ie.the equivalent of

```
-I. -I$AST_ROOT/kernel/lib/inc -I/star/include
```

on the command line. The strategy of C use in ASTERIX has been to restrict it to the kernel routines – there are only two exceptions, `fit__misc_c.c` in the spectral fitting and `htrace_int.c` in the HSD editor packages. If it became necessary to start adding more C routines then there might then be a case for a local `GEN_make` token containing local C include directives to give the same flexibility as is already implemented in the Fortran case.

In summary there are only two tokens in this directory class, namely **F_INCS** and **C_INCS**. The only built file is the Fortran mapping file which ends up in `$AST_ETC/sys/<subsystem>.includes` where *subsystem* is the full subsystem name of the include file diretory.

2.4.7 The `lib` Class

The `lib` class handles the construction of one or more object libraries

2.4.8 The `appgrp` Class

Handles the code associated with an ADAM application group.

2.4.9 The `module` Class

The module directory class supports both the `export` and `export_source` targets, just like the top level. This enables individual modules to be released as separate entities.

The mandatory tokens supported by the module class are **VERSION** and **DESCRIPTION**. The values of these tokens are used to provide information for the selection of modules at user ASTERIX startup. A module must supply a startup script for each supported shell (currently only `csh`). This script is `sourced` for each module that is to be initialised.

The module class obeys the **EXPORT** and **EXPORT_SOURCE** targets which construct binary+source and source only module releases respectively. The module release is written to a compressed `tar` file, the location of which is specified using the `EXPORT` environment variable, the default being the current directory, ie. the module top level directory. This facility can be used to release modules of ASTERIX separately of other modules.

2.4.10 The `master` Class

The master class is the template for the top level package. The only build function performed by such a directory is the creation of the system Tcl index file in `$AST_ETC/tclIndex`. This is done after building all the sub-directories which are all modules at this level.

The master class obeys the **EXPORT** and **EXPORT_SOURCE** targets which construct binary+source and source only releases respectively. The release is written to a compressed `tar` file, the location of which is specified using the `EXPORT` environment variable, the default being the current directory.

2.5 The `cmk` script

Now that the directory class structure has been introduced we can mention the `cmk` script without hopelessly confusing the reader. `cmk` takes two or more arguments. The first argument is the name of a directory class (eg. `script`, `appgrp` etc). Subsequent arguments are concatenated to form a command. `cmk` then effectively runs `mk` with that command *only* on directories whose class is the same as the first argument from the current directory downwards. For example,

```
% cd $AST_ROOT
```

```
% cmk scripts unbuild
```

Unbuilds every scripts directory in ASTERIX. This is an extremely useful thing to be able to do. If you have to make a change to the way the script directory class is handled, you can unbuild all the script directories, change the `.base_config` and/or `.make` files, reconstruct the `Makefiles` using `mk makes` then rebuild. Thus although the whole system is rather more fragmented then it used to be, giving a finer degree of control of parts of the system, there is also more opportunity for labour saving tricks.

3 Building a Release

To release ASTERIX we generate an exportable copy of the source code using the `export_source` top level target. We supply this file to Starlink who build it on each system. They check that the `install` and `deinstall` targets work ok, then they build the binary export version using `export`. It is this machine specific file which is released, along with a news item and update to SUN/98.

To test a release go through the same process, building ASTERIX on a scratch area. It is essential that you do the following,

- Make sure that the current directory specifier "." is not in your `PATH` variable before building the package. For security reasons most people work like this, and its is important that the ASTERIX make files do not make the assumption that this directory is in the path.
- On Solaris ensure that **LD_LIBRARY_PATH** is set correctly. It should be

  ```
  /star/lib:/usr/openwin/lib:/usr/ucblib:/opt/X11R5/lib:/star/share
  ```

  It absolutely must not contain anything to with XANADU or FTOOLS when you build ASTERIX, or dire consequences will ensue.
- Make sure that the environment variable `AST_SYS_DEV` has no value. You can force this using,

  ```
  unsetenv AST_SYS_DEV
  ```

  If you don't do this then the build system will look for the source code in the SCCS libraries, which of course won't exist.

4 Open Issues

This is a list of those things which haven't been finished in ASTERIX V2.

- The documentation directory. I have implemented this as a simple variant of the `etc` directory class, with the documents being copied into `$AST_ETC/docs`. Starlink doesn't really have a system for recognising internal package documents, so I propose we simply don't tell them about this.
- The help system.