# Software System Overview

# XXXX Astro-H

Version 0.3

February 18, 2012

**ISAS/ GSFC**
Greenbelt, Maryland

**Prepared by:** David B. Hon, James Peachey

# Table of Contents

# CHANGE RECORD PAGE (1 of 2)

| DOCUMENT TITLE: Astro-H Software System Overview | | | |
|---|---|---|---|
| ISSUE | DATE | PAGES AFFECTED | DESCRIPTION |
| Version 0.1 | February 10, 2012 | All | First draft |
| Version 0.2 | February 16, 2012 | Most | Merged Build & Install & Unit Test Guide |
| Version 0.3 | February 18, 2012 | Sections 2, 3, 4, and 5 | Fleshed out details. Corrected spelling errors |

# Introduction

## 1.1 Purpose

This document describes the approach and design for Astro-H calibration and analysis software as developed in Build 0 and planned for future Builds.

## 1.2 Applicable Documents

The current approach and design are based on requirements contained in the following documents:

astroh_SCT_software-20110923.ppt
astroh-analysis-system-use-case.ppt

# 2 Software System Design

## 2.1 Development Lifecycle

The development life cycle for the software is ***iterative*** in nature. The largest iteration is called a ***Build***. Each Build is six months in duration, with deliveries on the second Friday of February and August. Within each build there will be shorter iterations called ***Cycles***. Detailed plans for Builds and Cycles are still being developed, but in any case, development goals for each Build shall be established at the beginning of the Build, and each Build delivery shall establish a baseline for the next Build.

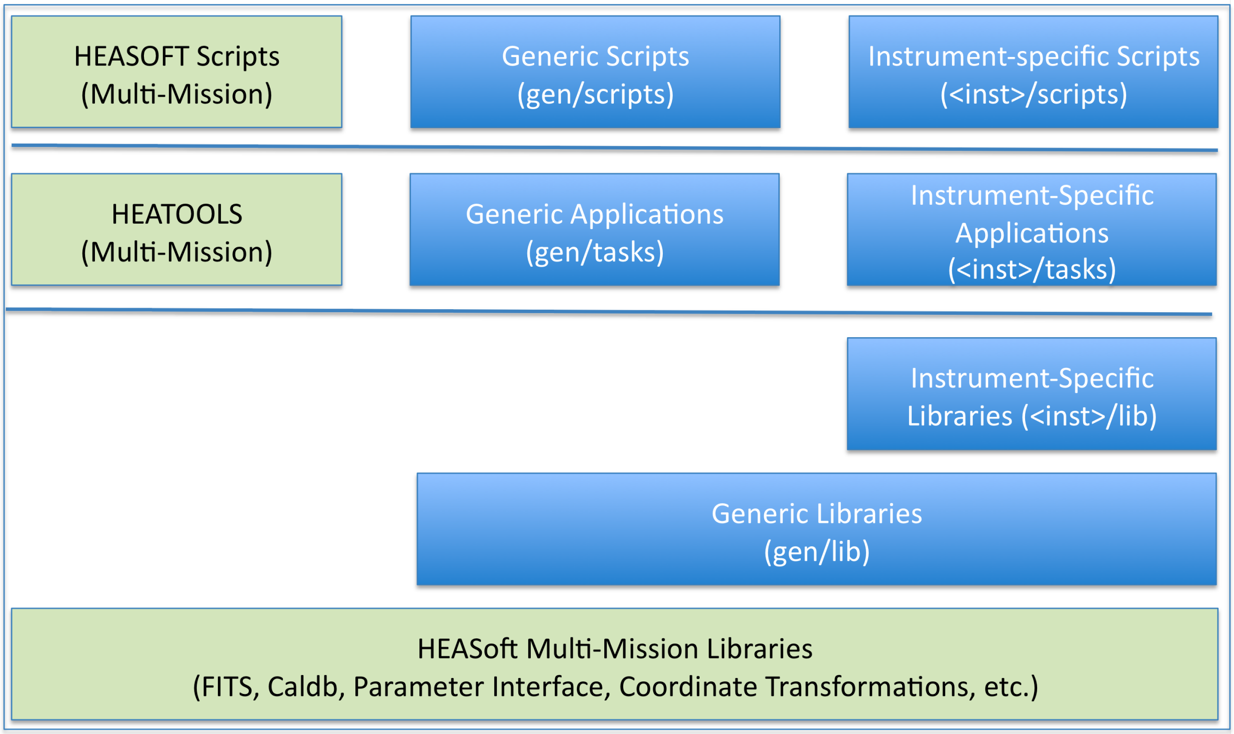
## 2.2 Design Overview and Rationale

The design of the software is a hybrid between a linear procedural design and an object-oriented (OO) design. This approach was adopted in order to meet the goals of simplicity, compliance with the HEASoft system, mission-independence, and minimized redundancy. A purely procedural design (in ANSI C) would require too much supporting code for custom data structures (complexity and redundancy). A full OO framework would add too much overhead and violate the goal of simplicity, requiring too much specialized knowledge for developers.
The current approach uses essentially intuitive procedural code that any developer should understand, but is in C++ and makes use of C++ exception handling, streams, and data structures in the C++ ISO standard template library. In addition, some classes may be developed in cases where an OO approach simplifies or reduces the code volume. However, such abstractions shall be kept small and specialized, without overuse of inheritance or templates.

## 2.3 Structural Architecture and Dependencies

The software is developed in layers as shown in Figure 1. The most fundamental (lower-level) elements are at the bottom of the diagram, with each subsequent layer including higher-level, more specialized functionality that in general builds on the layer below. Blue elements are Astor-H specific; light green elements are part of the HEASoft/HEADAS system. The lowest tier (below the bottom blue line) includes compiled libraries, the middle tier compiled applications, and the top tier (above the top blue line) scripts.

Figure 1. Architectural Layers




- **HEASoft Multi-Mission Libraries (HMML):** mission-independent and multi-mission components of the HEASoft software system: C and Perl libraries that support common data and user input functions. These libraries are developed and maintained by the HEASARC, separately from the Astro-H mission. Current components used by Astro-H include:
  - ape: parameter file access, located in the package "heacore" (C)
  - attitude: coordinate transformation library, located in the package "attitude" (C)
  - cfitsio: low-level FITS file access, located in the package "heacore" (C)
- **Generic Libraries:** Astro-H core C and Perl libraries that are not instrument-specific. These libraries shall be engineered to minimize the amount of mission-specific code, but this layer may contain Astro-H mission-specific functionality as needed. Components in this level may (and generally do) utilize components in the HMML. Current components include:
  - ahfits: higher-level FITS file access (C++)
  - ahgen: generic facilities to support a systematic, modular application design (C++)
  - ahlog: output and logging facilities (C++)
- **HEATOOLS:** multi-mission utilities for manipulating FITS files. These applications are maintained by the HEASARC, and utilize the HMML layer. Examples include:
  - ftdiff: tool that determines and reports differences between FITS files (C)

    - ftlist: tool that prints to the terminal content from a FITS file in textual form (C)
    - ftverify: tool that validates FITS files (C)
- **Generic Applications:** application code for Astro-H specific calibration and analysis tasks. These applications utilize code from the Generic Libraries layer and the HMML layer. These applications are meant to be a very thin layer on top of the Generic Libraries layer. As much as possible, the functionality in the Generic Applications is implemented in the Generic Library layer, following established interface constraints. Whenever possible, Generic Applications avoid calling functions from the HMML directly. Current applications include:
    - ahdemo: proof of concept template/sample tool that demonstrates a proposed design flow for event-file-oriented applications, and the use of the libraries (C++)
- **HEASOFT Scripts:** multi-mission script utilities for manipulating FITS files. These applications are maintained by the HEASARC, and utilize the HMML layer, and also applications from the HEATOOLS.
- **Instrument-Specific Libraries, Applications and Scripts:** application code for instrument-specific Astro-H specific calibration and analysis tasks. No current applications are implemented. Full specifications for these layers will be developed during Build 1.

## 2.4 Directory Layout

Astro-H software shall be laid out as a subdirectory of the HEASoft package, parallel to other current missions such as Swift and Suzaku, and HEASoft mission independent sub-packages, such as Heacore and Heatools.
Under the top directory are the following subdirectories:

- **BUILD_DIR:** standard directory for building and installing HEASoft software
- **doc:** documentation directory
- **gen:** generic Astro-H software, that is, not specific to any one instrument

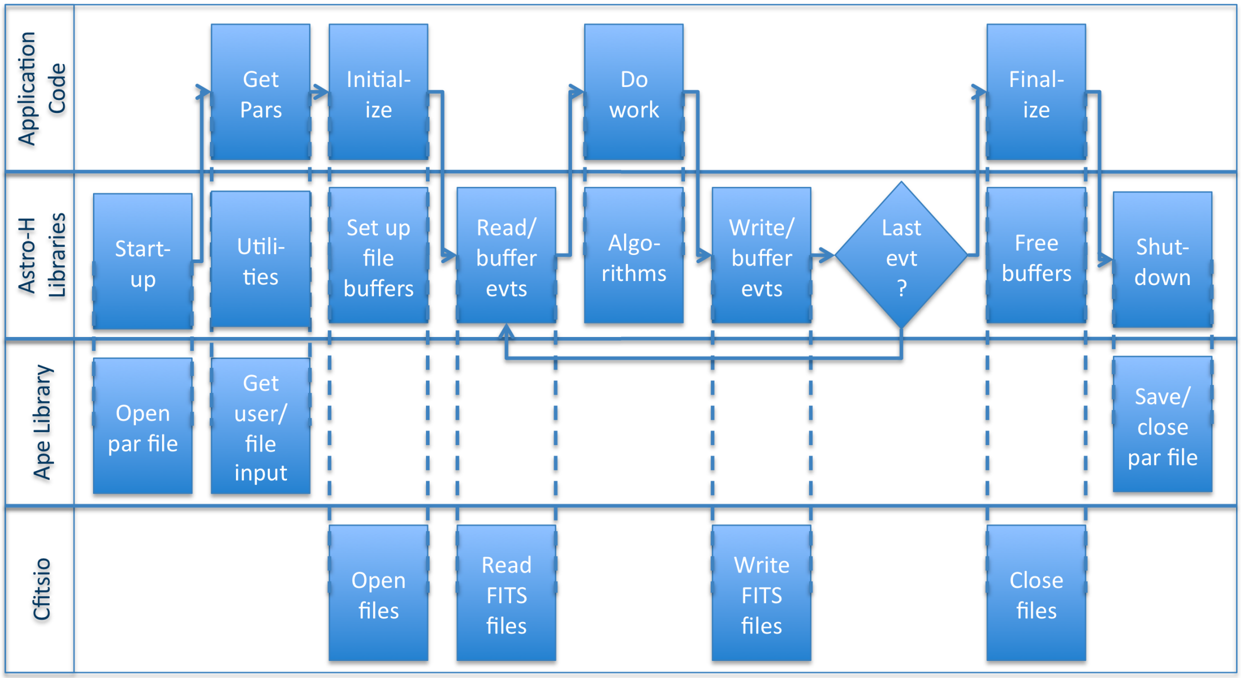
Additional directories parallel to gen for each instrument shall be added in the future, i.e., hxd, sxs, etc.
Under the gen directory are the following subdirectories:

- **aht:** contains all files associated with the Aht generic testing tool
- **lib:** contains a subdirectory for each (generic) library
- **tasks:** contains a subdirectory for each (generic) application
- **ut:** contains directories with unit test input data, aht manifest files, etc. (not fully populated in Build 0)

## 2.5 Application Flow Patterns

In general, applications shall strive for a simple, procedural structure, following the HEASoft paradigm of "ballistic" software: following initial user input, the tools run to completion without further user interaction. It is envisioned that most applications will follow one of a small number of ***Application Flow Patterns*** such as the one shown in Figure 2.

Figure 2. Event-oriented Application Flow Pattern




This pattern is closely related to the Suzaku ANL tool architecture (Start-up = ANL startup, Get Pars = ANL com, Initialize = ANL init, etc.) Unlike ANL, which is a framework, the main application execution flow is controlled by a procedural "main" function in the application code. Key steps in this pattern are:

- **Start-up:** global/universal set-up: open log files (not shown), parameter file, etc.
- **Get Pars:** application-specific input parameters
- **Initialize:** set up/connect FITS files to buffers for columns of interest to application
- **Read/buffer Events:** read the data of interest to the application
- **Do Work:** application-specific operations, using algorithms from the libraries
- **Write/buffer Events:** write data of interest to output
- **Finalize:** close files and free resources in parallel to the "Initialize" step
- **Shutdown:** global/universal cleanup: close/save parameter file, log files, etc.

# 3 Guide To Building and Installing the Release

Some prior experience with the UNIX/Linux shell command-line -- (t)csh or (ba)sh, Perl, and HEASoft/HEADAS Ftool development in C and/or C++ is assumed. Also please note that this release has been developed and tested with Scientific Linux 5.x and 6.x and Mac OSX. The versions of Perl that have been used for the Perl modules developed for the 'aht' script include 5.8, 5.10, and 5.14 (so far).

## 3.1 Fetch the distribution file or perform a CVS Checkout of Tag AstroH_B00

If one has direct access to the HEADAS and AstroH CVS repository, one may extract the current build via the CVS pserver at Goddard. For example, assuming one's current working directory is $HOME:

```
# this is usually a one-time login:
cvs -d :pserver:${USER}@daria:/headas login
#
cvs -d :pserver:${USER}@daria:/headas co -r AstroH_B00 -d b00astroh headas
pushd b00astroh
cvs -d :pserver:${USER}@daria:/astroh co -r AstroH_B00 astroh
```

The first CVS checkout above should create a new subdirectory 'b00astroh', which one should then cd or pushd into to perform the second CVS checkout. The second CVS checkout will create the 'astroh' sub-directory.

If CVS access is not available, there is a distribution file available for 30 days as of February 10, 2012:
https://webdrive.gsfc.nasa.gov/longauth/600/david.b.hon/1o7uGoC
Please use the following to login:
username = astroh-build0
password = astroh-build0

### 3.2 Build the Release (AstroH_B00)

The AstroH software leverages the standard HEADAS environment and thus requires a prior installation of a recent HEASoft release. Those who have direct access the HEADAS CVS repository may follow the instructions below. For those who lack direct access to the CVS, please first download and install the latest HEASoft release, then build AstroH software from the tar or zip file (need URL link for download).
Building the runtime and invoking the applications hinges on proper setting of the essential environment variable $HEADAS. The $HEADAS environment variable setting is directly related to the configured installation 'prefix', as described below. Before performing the AstroH specific build, one must 'source' an existing HEADAS install script.

In (t)csh:  source $HEADAS/headas-init.csh

In (ba)sh: . $HEADAS/headas-init.csh

Assuming one has configured a Makefile to install the HEASoft build into the prefix $HOME/local/astrohb00, a Linux shell environment variable $HEADAS should look something like this:

$HOME/local/astrohb00/x86_64-redhat-linux-gnu-libc2.12
or
$HOME/local/astrohb00/x86_64-redhat-linux-gnu-libc2.12

The system architecture text appended to the configure installed prefix is can be suggested by the 'Target' line output of 'gcc -v' and a directory listing of '/lib/libc-*'.

### 3.3 Configure and Hmake to Compile and Install

Regardless of whether one has performed a CVS checkout, or a tar-file has been downloaded and extracted, the next step is to perform the Makefile configuration within the top-level BUILD_DIR. That is, assuming one has a fresh extraction directory '$HOME/b00astroh', cd or pushd there and in (t)csh or (ba)sh:

```
#
pushd BUILD_DIR
# use this intermediate environment variable for convenience:
# this becomes our new HEADAS installation directory...
# (ba)sh:
# export HEA=~/local/astrohb00
# or (t)csh:
# setenv HEA ~/local/astrohb00
#
./configure --prefix=$HEA --enable-symbols –with-components='astroh heatools'
hmake
hmake install
hmake test
hmake install-test
#
# now reset HEADAS and setup the env. of the freshly installed system:
export HEADAS=${HEA}/x86_64-unknown-linux-gnu-libc2.12
#
echo "HEADAS == ${HEADAS}"
# (ba)sh:
# . ${HEADAS}/headas-init.sh
# or (t)csh:
# source ${HEADAS}/headas-init.csh
#
echo sourced new ${HEADAS}/headas-init.sh
# note for (t)csh a rehash may be needed...
```

Please note that the source code directory location $HOME/b00astroh is separate and distinct from the HEADAS / HEASoft installation directory $HEADAS == $HOME/local/astrohb00.

Note that $HEADAS is a fundamental environment variable required for the built and for runtime apps., while $HEA is just a convenience for the build. Also note the single quoted, space separated list of component elements to build – this may change for future builds.

### 3.4 Snapshot of Hmake Installation

If the build is successful, the following directory listing:

ls -l $HEADAS/bin | grep astroh

should yield:

```
ahdemo -> ../../astroh/x86_64-unknown-linux-gnu-libc2.12/bin/ahdemo*
aht -> ../../astroh/x86_64-unknown-linux-gnu-libc2.12/bin/aht*
testahfits -> ../../astroh/x86_64-unknown-linux-gnu-libc2.12/bin/testahfits*
testahgen -> ../../astroh/x86_64-unknown-linux-gnu-libc2.12/bin/testahgen*
testahlog -> ../../astroh/x86_64-unknown-linux-gnu-libc2.12/bin/testahlog*
```

and

ls -l $HEADAS/bin  | egrep 'help|diff'

should yield:

fhelp -> ../../heatools/x86_64-unknown-linux-gnu-libc2.12/bin/fhelp*
ftdiff -> ../../heatools/x86_64-unknown-linux-gnu-libc2.12/bin/ftdiff*
fthelp -> ../../heatools/x86_64-unknown-linux-gnu-libc2.12/bin/fhelp*

After sourcing the freshly installed headas-init script:

Fhelp should be available for the astroh apps. under the 'bin/' directory.

Ftdiff is essential for the unit test final validation step(s).

Perform a 'which aht' and check that the AstroH unit test Perl script is found in one's PATH.

# 4 Testing Standards

## 4.1 Compiled Unit Tests

A pattern for unit tests for compiled code is being developed and will be formalized in Build 0. In any case, Test-Driven-Development techniques shall be employed to ensure robust and comprehensive unit testing of libraries and applications as they are developed.

## 4.2 Application Unit Tests

The result of the build installation is a set of UNIX/Linux command-line applications, directly invokable from one's login shell. Check that 'fhelp ahdemo' and 'fhelp aht' display something useful, and please read.
The installation listing above shows a handful of binary applications and the Perl script 'aht'. Generally a HEASoft user need only set their runtime $PATH shell environment variable to point the $HEADAS installed binaries, and another environment variable $PFILES must be set to support various Ftool command-line options. However, when developing and testing (new or not) apps., it is essential to ensure that a specific set of environment variable settings are used.

## 4.3 Initialization of the Unit Test Manifest, Directories, and Runtime

The Perl script 'aht' installed by the hmake provides a kick-start. It can be used, in theory, at any point in the software development cycle. Assume one creates an new empty directory under ones' $HOME and runs aht the very first time from within that directory:
mkdir $HOME/utest ; pushd $HOME/utest
then:
aht -i
or
aht --init
should initialize the AstroH unit test (aht) sub-directories and create an initial version of the unit test manifest and shell environment setup script(s). Here is a partial directory listing of the results (ls -l ./ ./log ./etc):
./:

aht_manifest.pl
etc/
expected_output/
input/
log/
output -> ./xen.gsfc.nasa.gov/2012Feb09/14.51.23/output/
xen.gsfc.nasa.gov/
./log:
./log/aht_stderr2012FebDD@HH.MM.SS
./log/aht_stdout2012FebDD@HH.MM.SS
./etc:
./etc/setup.csh
./etc/setup.sh

### 4.4 The Unit Test Manifest (./aht_manifest)

Please note the directory listing above shows the file 'aht_manifest.pl'. This text file is used to instruct the unit test execution, performed by the aht Perl script 'aht -t' invocation ('fhelp aht' should provide command-line help). The manifest file provides a description of the desired unit test that includes the name of the AstroH Ftool to be tested, its default command-line options, its inputs and expected outputs, and other essential information about the test.

The manifest may be hand edited and also modified by the 'update' option of the aht script ('aht -u'). The manifest is itself a Perl script, so please be careful about hand editing it. The initial version of the manifest is simply created by a Perl 'Here' statement, and the build 0.0 version explicitly indicates that the AstroH Ftool meant to be tested via this initial manifest is 'ahdemo'.

### 4.5 The Shell Environment Variable Setup (./etc/setup.sh)

The directory listing also shows two shell script files under the ./etc sub-directory. They should be congruent, but the (ba)sh is critical to the unit test "harness". While a user may prefer to use (t)csh at login and in general, the Perl runtime is "hard-coded" to use (ba)sh when it executes an external application (our Ftool to be tested) as a child processes.

A unit test should be conducted in a controlled fashion within a specific runtime environment. The ./etc/setup.sh shell script provides the runtime environment by ensuring the standard $HEADAS and $PFILES environment variables (and any others required) are set. Each unit test of an AstroH Ftool application that is conducted via the aht Perl script 'spawns' a child process within a sub-shell whose runtime environment is explicitly established by the content of './etc/setup.sh'.

The AstroH developer (and tester) must ensure the integrity of the test either by manually setting their environment variables beforehand, or by editing the ./etc/setup.sh script appropriately. The current version (B0 release) of the setup script does not override the user's existing, environment variable values, setting them only if not currently set. Upon invocation of 'aht -t' the ./etc/setup.sh is effectively "sourced" before the child process is "spawned". Please note that the manifest file provides an entry that indicates the location of the setup.sh, so in principle a setup.sh script may be placed elsewhere (other than ./etc), and potentially shared across multiple test manifests.

In addition to ensuring that required environment variables are present, the setup shell script also performs an "ulimit". The ulimit simply sets various process runtime limits like maximum file size, memory use, etc. to the system's allowed max. One particularly interesting process limit is the max. "coredumpsize". It is important that the "coredumpsize" be non-zero, otherwise Perl will not be able to detect if a child process\ (our Ftool) experiences a fatal segmentation fault or some other error/exception it (the Ftool) fails to handle properly. The developer should manually perform either a (t)csh "unlimit" or a (ba)sh "ulimit" to determine if there system coredumpsize is non-zero. If it is, a system admin. may be needed to enable non-zero core dumps.

### 4.6 Establishing Test Inputs

The (aht -i) initialization creates an empty ./input sub-directory. While it may be useful to conduct a unit test with ill-defined inputs to verify the Ftool's exception handling, ultimately one must provide some reasonable set(s) of test input files. The test input files may be copied directly into ./input, or symbolic links within ./input may be created.
The manifest file provides an entry to list input files (as a Perl hash key => value statement). The developer may manually edit this entry. Alternately, the update invocation of the unit test harness script (aht -u) will check the directory contents of ./input and insert the result into the manifest. A view of the updated manifest file should show the new entry setting/value(s), as well as any prior entries commented-out (via Perl #).

For convenience, a very small set of input test FITS files have been placed into the AstroH CVS repository to support regression tests of the build 0 components. For future builds, however, we expect a considerably large set of test input data files. Consequently use of CVS repository will be discouraged for test data files, and some other repository mechanism must be established -- and perhaps the test harness will be enhanced to fetch test (validated) data inputs form the indicated data repository.

In addition to input data files, an AstroH Ftool application will also need an associated text file that provides the standard set of Ftool runtime parameters and conforms to standard HEASoft "parfile" text file format. The build installs default "parfiles" into the $HEADAS/syspfiles directory. When invoking/testing an Ftool application, a user may be prompted (depending on the nature of the command-line invocation) to type in values that will then be used to create or update a parameter file in the user's $HOME/pfiles sub-directory. By HEASoft convention, pfile filenames are expected to match the Ftool binary application name, with a ".par" extension. For example, our AstroH build 0 "ahdemo" Ftool application has a pfile filename "ahdemo.par". The developer and/or tester may manually copy a pfile to the local ./input subdirectory, but if one is not present, 'aht -i or -u or -t" will attempt to copy one from $HOME/pfiles or from $HEADAS/syspfiles. Please note that whenever a test is executed by 'aht -t', the $PFILES environment variable is reset to './output;./input' -- to ensure the self-contained and controlled nature of the test.

### 4.7 Executing Unit Test Scenarios and Updates

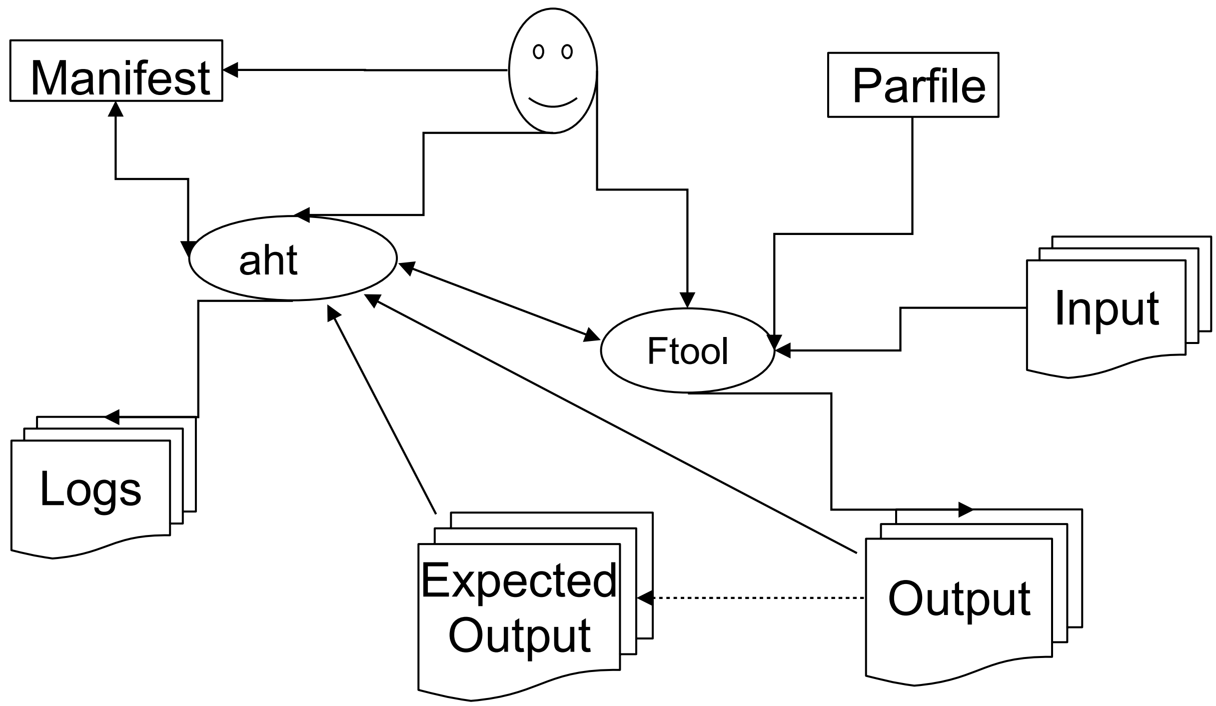
Once input(s) are established, the Ftool should be executed either manually or via the test harness (aht -t). Whenever 'aht -t' is used to run the Ftool, a new output sub-directory is created and a symbolic link to it is established. When the developer is satisfied with the results of the Ftool output, the 'aht -u' update should be invoked to scan the most recent output subdirectory content and copy the files found into the ./expected_output sub-directory -- this also updates the contents of the manifest file.

Short of running a specific test, one may use aht command-line options to first verify the expected runtime behavior. For example, 'aht -e or --env' should process the setup.sh script and print the results to stdout. And/or one may invoke 'aht -p or -par' to print the current parfile contents. And/or one may invoke 'aht -n or --noop' to see a printout of everything that will happen short if actually executing the Ftool as a child proc. Some of the options accept an optional value, for example 'aht -u foo' will update the manifest by changing the Ftool application name to "foo", ditto for 'aht -t foo' -- which will also proceed to attempt a test of the 'foo' binary application Another option short of falling executing the test is 'aht -d or --debug', which simply invokes the GNU debugger 'gdb. More information about 'aht' command-line options is available via 'fhelp aht'.

The standard Perl POSIX module provides a Perl script with the ability to establish a signal handler and check that a child process is "alive" and also evaluate the exit status of the child. Once the aht test script successfully "spawns" the Ftool child proc., it enters an "event" loop that iterates over the redirection of the child's stdout and stderr to the current ./output sub-directory, while checking that the child proc. is actively running. The unit test manifest file provides an entry that allows the developer or tester to establish some notion of how long the child process should run, via a timeout value (in seconds). The event loop decrements a timeout counter, and allows the user to either interrupt and terminate a test via control-C. If the timeout limit is reached before the child exits, the user is prompted to continue or terminate the test -- presumably due to some unexpected behavior of the Ftool application However there may also be some unexpected behavior of Perl's signal handling and child exit status evaluation, depending on the version of Perl used (we need to do a bit more evaluation of this).

### 4.8 Unit Test Context Diagram and Sample Invocation

The context diagram below provides an overview of the aht script activities:




Here is a brief recapitulation of the Ftool development and unit test activities:

1. Astro-H FITS tool (Ftool) application developer establishes test input and develops and runs the Ftool application manually to produce expected output(s).
2. A unit test manifest file is initialized and optionally hand edited to an indicate he nature and specifics of a test.
3. The Astro-H test (Perl) script 'aht -t' is invoked, perhaps a number of times, to execute the Ftool application as a child proc.
4. The Astro-H test (Perl) script 'aht -u' is invoked to update the manifest file and the contents of expected output(s) (via a copy of successful output).
5. All Ftool printout to stdout and stderr are redirected by aht.pl to text log files.
6. The user may interrupt an 'aht -t' test via a Control-C (interrupt signal).
7. The aht Perl script monitors signals to determine if the Ftool child proc. completes execution successfully or fails, and reports its exit value.
8. Upon a successful exit value, aht.pl compares the output(s) with what is expected.
9. Upon unsuccessful exit value, aht.pl searches logs for keywords (described in the manifest file) and reports accordingly.
10. Proceed to another test.

For example, first invoke ahdemo manually (after putting some input into ./input):

ahdemo input/event_file_1.fits output/outfile.fits templatefile = input/sff_mockup_template.tpl clobber = Y

Then initialize and update the full unit test setup:

aht -i
aht -u
And run the test harness:
aht -t

Below is a snapshot of the sub-directories populated by the build 0.x unit test of 'ahdemo' (an edited, partial directory listing via ls -R ./). Note that one backup version of the manifest is preserved as a "hidden" file (in the event a manual edit needs to be undone).

```
./:
.aht_manifest.pl
aht_manifest.pl
etc/
expected_output/
input/
log/
output -> ./xen.gsfc.nasa.gov/2012FebDD/HH.MM.SS/output/
xen.gsfc.nasa.gov/

./etc:
setup.csh
setup.sh

./expected_output:
ahdemo.par
ftdiff.par
outfile.fits
stderr/
stdout/

./expected_output/stderr:
ahdemo

./expected_output/stdout:
ahdemo

./input:
ahdemo.par
event_file_1.fits -> ../../tasks/ahdemo/input/event_file_1.fits
event_file_2.fits -> ../../tasks/ahdemo/input/event_file_2.fits
event_file_3.fits -> ../../tasks/ahdemo/input/event_file_3.fits
ftdiff.par
sff_mockup_template.tpl -> ../../tasks/ahdemo/input/sff_mockup_template.tpl


./log:
ahdemo_aht_stderr2012FebDD@HH.MM.SS

./xen.gsfc.nasa.gov/2012FebDD/HH.MM.SS/input:
ahdemo.par
event_file_1.fits
event_file_2.fits
event_file_3.fits
```

ftdiff.par
sff_mockup_template.tpl

./xen.gsfc.nasa.gov/2012FebDD/HH.MM.SS/output:
ahdemo.par
ftdiff.par
outfile.fits
stderr/
stdout/

./xen.gsfc.nasa.gov/2012FebDD/HH.MM.SS/output/stderr:
ahdemo

./xen.gsfc.nasa.gov/2012FebDD/HH.MM.SS/output/stdout:
ahdemo

### 4.9 Executing A Set of Unit Test Scenarios

- There should ultimately be provisions for conducting multiple unit tests that may share test data input. The current implementation of the unit test harness manifest file and Perl modules provide the beginnings of such. The notion of 'sets of unit tests', however, is not yet fully implemented. For build 0, there must be a separate unique manifest file for each individual unit test. Furthermore the focus of the aht Perl script, when conducting a test, is currently a hard-coded manifest file name; and the file is expected to be present in the current working directory with the specific path-and-file-name './aht_manifest.pl'. Consequently when the developer 'finalizes' the unit tests of an Ftool application and provides a set of individual manifest files, the tester must indicate which unit test to conduct by setting a symbolic link:
- ln -s path-to-manifest-file/manifest-filename.pl ./aht_manifest.pl
- Future versions of the aht Perl script may provide a command-line option 'aht --manifest filename'. Future versions of the manifest file content may provide further description of the unit test set. Or perhaps a new sub-directory of './manifests' that contains the complete set manifest files may be sequentially processed by the unit test script, etc.

### 4.10 Thread Tests

Starting in Build 1, tests that thread together multiple tools shall be developed in order to test pipelines, subsystems, or analysis threads. Facilities shall be provided that build on aht to provide this level of testing support.

## 5 Build 0 Implementation

### 5.1 Libraries

The following libraries are implemented in Build 0:

- **libahlog:** library containing logging and output facilities, which allow text output, warnings, informational messages and errors to be directed to the screen and/or to files.

- etc.