SOFTWARE SYSTEM OVERVIEW
XXXX Astro-H

Version 0.9

April 26, 2012

ISAS/ GSFC
Greenbelt, Maryland

Prepared by: David B. Hon, James Peachey

Table of Contents

CHANGE RECORD PAGE

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
Version 0.4	03/08/12	Section 3	CVS and Build
Version 0.4a	03/13/12	Section 3, 4, and Appendix M	Feedback edits of CVS and Build and Test, and Appendix M (Manifest-file example)
Version 0.5	March 12, 2012	New Section 4	Coding Standards
Version 0.6	March 26, 2012	New Section for ahlog; small correction to cvs instructions.	Initial description of ahlog
Version 0.7	April 23, 2012		Begin to describe doxygen tags
Version 0.8	April 26, 2012	Various sections	With mark-up from working meeting
Version 0.9	April 26, 2012	Page 1	Update revision number and date

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.

James 4/27/12 9:41 AM

Comment: Lorella to write larger introduction:
1. Principles
2. Requirements for specifying algorithms, parameters etc.
3. How we will do things, e.g., how we will use doxygen

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

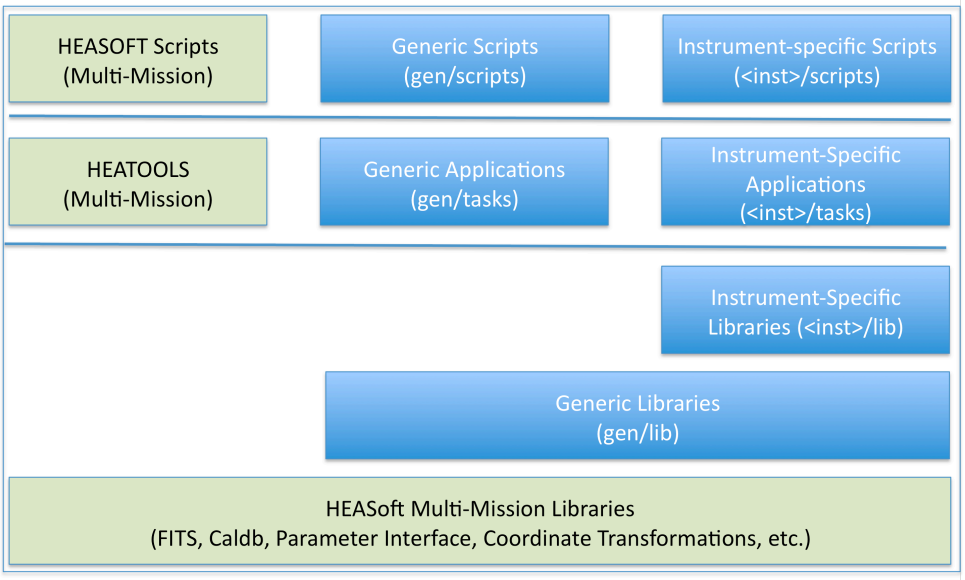
The design of the software is a hybrid between a linear procedural design and an object-oriented (OO) design. This approach was adopted in order to meet the goals of simplicity, compliance with the HEASoft system, mission-independence, and minimized redundancy. A purely procedural design (in ANSI C) 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

James 4/27/12 10:49 AM

Comment: Lorella: show tree of directory from presentation.
Lorella to discuss naming conventions for tasks.

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

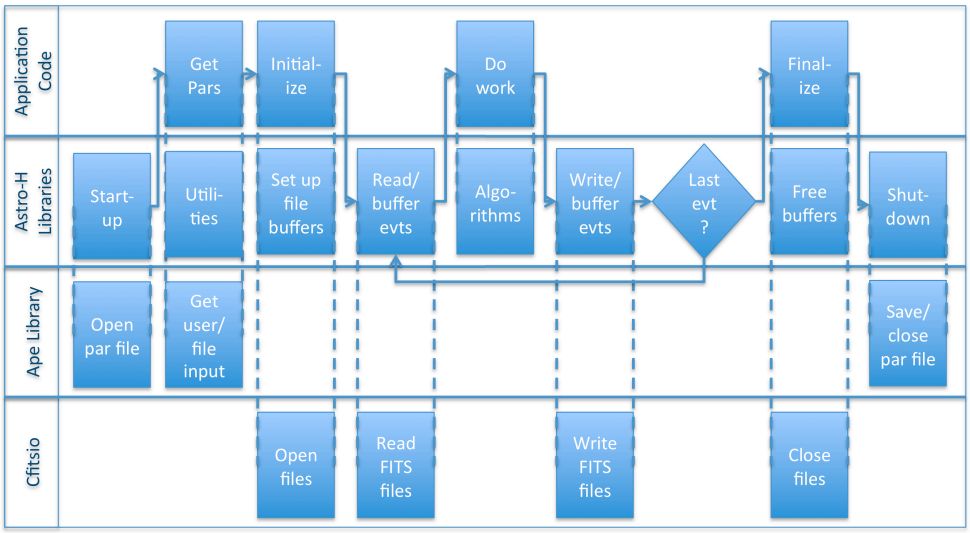
Additional directories parallel to gen for each instrument shall be added in the future, i.e., hxd, sxs, etc.
Under the gen directory are the following subdirectories:

- **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 (Build 0) 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

The Astro-H software repository is hosted at Goddard Space Flight Center using Concurrent Versions System (CVS see http://en.wikipedia.org/wiki/Concurrent_Versions_System). 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 more information about CVS and its pserver, please refer to the manual (online: http://ximbiot.com/cvs/manual/cvs/1.11.23/cvs_2.html#SEC29).

The HEADAS CVS contains multiple repositories, one for each HEA mission and one for core/common libraries and Ftools. Access via the CVS pserver to each repository is controlled by an Access Control List (ACL). One must obtain a unique CVS pserver account and establish a password. The account name need not be congruent with one's host system account, and the password must differ from one's host login. Although a CVS pserver user needs only one account, the nature of the ACLs requires a separate pserver login for each repository. Each repository's ACL lists which user accounts have either read-only or read-and-write access to that specific repository.

To access the Astro-H CVS repository, please request a pserver account and initialize its password by sending email to system@milkyway.gsfc.nasa.gov . One's (pserver) account name is must added to two ACLs – one for the "headas" core repository, and another for the "astroh" repository, so be sure to request access to both. Also, please provide the IP address for each host that will connect to the CVS pserver client. Generally a pserver client need only perform a one-time login from each host machine, and the CVS client (Linux command-line "cvs") will maintain a login info. On UNIX/Linux systems this is a file in one's host home directory: HOME/.cvspass. If the file is renames of removed, one will need perform another login.

Building the Astro-H software from source requires code from two CVS repositories: "headas" and "astro", and a separate pserver login for each. For example, assuming one's current working directory is $HOME, and also one's CVS pserver account name is congruent with one's host account name (which may not be the case) one may perform a CVS pserver login as shown in the following. Also please note, for convenience, the smaller font text below may be cut-n-pasted into shell scripts.:

```
# this is usually a one-time login for each host, for each repository
cvs -d :pserver:${USER}@daria.gsfc.nasa.gov:/headas login
# and type the password at the prompt
#
# and also login to astroh (same account name and password):
cvs -d pserver:${USER}@daria.gsfc.nasa.gov:/astroh login
# and type the password at the prompt
#
# then proceed to checkout the source code...
```

An internal symbolic link has been established within the "headas" CVS repository that points to the "astroh" repository, so the checkout (co) below will also perform an Astro-H checkout (assuming one has logged into both repositories!). The CVS checkout command below should create a new directory 'b00astroh' which contains the headas source, and the 'astroh' sub-directory underneath.

```
# the following performs a recursive (all sub-directories) source code checkout of all elements in the HEADAS and
# Astro-H repository that have been tagged (labeled) for the Build 0 release via " AstroH_B00".
#
cvs -d :pserver:${USER}@daria.gsfc.nasa.gov:/headas co -r AstroH_B00 astroh
#
# the above "cvs co" command creates and populates the directory (named via the 2nd "-d" command-line option,
```

with the revision tag (the "-r" option), via the CVS pserver on daria, everything in the "/headas root repository"
associated with the module name "headas" (he final command-line arg.).

The above checked-out "headas" module should be self contained and self-consistent, and include all "heacore", "heatools", and "astroh" elements. Please note that without the first "-d" command-line option used above, the cvs client application will check for the existence and value of the standard environment variable $CVSROOT.
If $CVSROOT is set to "pserver:${USER}@daria.gsfc.nasa.gov:/headas", (and furthermore if another environment variable $CVS_RSH is set to "ssh"?) the first "-d" option may be omitted.

3.2 Build the Release (AstroH_B00): Configure and Hmake to Compile and Install

The AstroH software leverages the standard HEADAS environment. 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 and installing HEADAS and Astro-H follows the established GNU-like pattern of "configure; make; make install". By default, however, configuring the make without specifying the installation destination directory (i.e. without –prefix), will NOT install into /usr/local, rather into a sub-directory under the user's current working directory. We designate this default build configuration as simply a "local private install". Configuring the build with "--prefix" to install into a more public location we designate as a "system public install".

Building a system public installation yields a set of runtime libraries and applications whose usage 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. After installing an AstroH specific build, one must 'source' the resulting HEADAS install script in order to access the installed apps. and runtime libraries.

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

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

Assuming one has configured a Makefile to install the Astro-H build $HOME/local/astrohb00, a Linux shell environment variable $HEADAS should look something like $HOME/local/astrohb00/arch-OS-libc, for example:

$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-*'.

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 checkout extraction directory $HOME/b00astroh (or ~/b00astroh), "cd" or "pushd" there and in (t)csh or (ba)sh proceed to configure and make "local private" or "system public" installations. The simplest installation (local private) configuration is described first:

```
# “cd” or “pushd” to the CVS checkout directory created earlier, into its sub-directory “BUILD_DIR”:
#
pushd $HOME/b00astroh/BUILD_DIR
#
# to build a local private installation, simply use the default configuration, – we recommend compiling with
# debugging symbols:
#
./configure –enable-symbols
#
# if the above succeeds, proceed to compile the libs and apps. via “hmake” – a simple wrapper of “make”,
# using the standard “make target” invocation:
#
./hmake
./hmake test
./hmake install
./hmake install-test
```

The simplest installation (local private) configuration described above yields the new directory within the checkout directory: $HOME/b00astroh/arch-OS-libc, for example: $HOME/local/astrohb00/x86_64-redhat-linux-gnu-libc2.12

An installation in an area other than the CVS checkout directory one may wish to share with other users (system public) is accommodated by the additional configure command-line option “--prefix”, described next. Note that an installation into “/usr/local” generally must be performed via the system administration account “root”:

```
#
# if one wishes to build a system public installation,  it is convenient to use the $HEA intermediate environment
# variable: this becomes our new HEADAS installation directory, and the HEADAS environment variable
# must be reset to point to it.
#
# (ba)sh:
export HEA=~/local/astrohb00
# or as root:
export HEA=/usr/local/astrohb00
#
# or (t)csh:
setenv HEA ~/local/astrohb00
# or as root:
setenv HEA /usr/local/astrohb00
#
# and make sure all users have access:
chmod a+rx $HEA
#
./configure --prefix=$HEA --enable-symbols
#
# if the above succeeds, proceed to compile the libs and apps. via “hmake”
#
./hmake
./hmake test
./hmake install
./hmake install-test
#
# please note the host architecture and OS and C library version information as formulated in the resulting
```

```
# installation directory and reset the HEADAS env. variable and setup the runtime of the freshly installed system.
#
# below is a an example setting for a 64 bit generic Linux host...
#
export HEADAS=${HEA}/x86_64-unknown-linux-gnu-libc2.12
# or (t)csh:
setenv HEADAS ${HEA}/x86_64-unknown-linux-gnu-libc2.12
#
echo "HEADAS == ${HEADAS}"
#
# now run / source the HEADAS runtime environment setup script:
#
# (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 $HEADAS is a fundamental environment variable required for the built libraries and for runtime apps., while $HEA is just a convenience for the build. Also, please note that the above example our source code directory location $HOME/b00astroh is separate and distinct from the system public HEADAS / HEASoft installation directory $HEADAS == $HOME/local/astrohb00, as a consequence of the "--prefix" specification used in the configuration script invocation.

3.3 Snapshot of Hmake Installation

If the build is successful, and one has reset $HEADAS to point to the new installation and "sourced" its "headas-init" shell script, 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 Coding Standards

4.1 Languages and Compilers

Compiled code shall be written in C++, conforming to the ISO standard (14882:2003) whenever possible. Any usage of code that does not comply with the ISO C++ standard shall first be approved by the Lead Developer. However, code must also compile with a set of current supported compilers, which shall be listed. Features included in the ISO standard that are not supported by all currently supported compilers shall not be used.
Scripts shall be written in standard Perl and shall use only Perl modules included with a vanilla installation of Perl. A list of currently supported versions of Perl shall be maintained. Scripts shall not use features that are not supported in all currently supported versions of Perl.

4.2 C/C++ Features and Usage

While the code is written in C++ and uses some language features, the code shall be fundamentally more like C, using only elements common to both languages, with the following exceptions:

- **Namespaces**: all library code that is part of the API shall use namespaces to encapsulate all code. The namespace shall match the library name, e.g., ahlog.
- **C++ Exceptions**: code shall use C++ exceptions for all exception handling. In cases where Astro-H code calls C code (extern "C"), e.g., cfitsio, the Astro-H code shall use the error handling protocol required by the code being called, ***but*** shall throw exceptions to report its own errors and unexpected results. Any new exception classes defined by Astro-H code shall inherit from a base class in the std::exception hierarchy.
- **Standard Templates**: code may use classes such as std::vector, std::map, etc. from the template library included in the ISO standard.
- **Streams**: output shall be performed using C++ streams. However, standard error messages, warnings, and output shall use facilities from ahlog, rather than directly using streams such as std::cout. Input and output from text files may be performed using C or C++ style functions. Input and output from FITS files shall use the ahfits library exclusively. In turn, ahfits shall use only cfitsio for lower-level FITS access.
- **Default Values**: default values in lists of arguments to functions or methods shall be provided (where appropriate) in header files, rather than in the source files.

4.3 Code-Level Documentation

All code shall be documented inline using Doxygen formatting (www.doxygen.org). Prior to releases, the full set of doxygen documentation shall be generated and included with the distributed code. One or more Doxyfiles (doxygen configuration files) shall be maintained in order to provide one or more levels of developer documentation. Doxygen tags used shall include:

- **file**: the name of each file (source and headers); used at top of each file

- **\author**: the name of the original author of each file (source and headers); used right below \file.
- **\date**: the date of the initial version of the file (source and headers); used right below \author.
- **\class**: the name of each class; used right before each class declaration.
- **\brief**: short summary of each method/function; used right before each function is declared (not where it is implemented).
- callgraph

5 Configuration Management Plan

5.1 Code Revision Control

James 4/27/12 10:29 AM

Comment: James to review and update this whole section as needed for current tagging conventions.

Developers shall have unrestricted read and write access to the Astro-H CVS code repository, as described in section 3. Developers are free to check in code as it is developed. Each check-in shall include a message about what the check-in accomplishes, in sufficient detail that someone other than the developer making the change will be able to understand the change at some later point. Developers are responsible for testing and integrating code.

5.2 Development Baseline: astrohdev

A baseline shall be established containing the latest testable configuration of Astro-H software. This baseline is called the Astro-H ***Development Baseline (astrohdev)***. The content of this baseline will change over time. To qualify for this baseline, the code must be ***complete***, meaning that the code compiles. Application code must also link to be considered complete. Code in the astrohdev baseline shall also be ***testable***, meaning that there is test code that exercises the functionality. This test code shall also be ***complete*** as defined above, and shall be run as part of the Makefile target "run-test" for the given component. When possible, astrohdev code shall also pass its tests.

5.3 Release Candidate Baseline

Prior to integration of software for each release, a ***Release Candidate Baseline*** shall be established containing the base revision of headas and Astro-H software proposed for that release. This baseline shall be named according to a convention that is yet to be determined. The content of this baseline ***shall not*** change over time. To qualify for this baseline, the code must meet all criteria for the Development Baseline. In addition, the integrator must verify that ***all*** Astro-H code with the given baseline passes all current automated tests, that is, tests that are run by the Makefile run-test target. A typical sequence for establishing a Release Candidate Baseline would be:

```
cvs -d :pserver:${USER}@daria.gsfc.nasa.gov:/headas checkout -r <headas-base-release> astroh_base
cvs -d :pserver:${USER}@daria.gsfc.nasa.gov:/astroh checkout -r astrohdev astroh
#
# Build the code and run the automated tests. If they pass, then at the top level execute the following:
cd headas
cvs tag <release-candidate-baseline-identifier>
```

where <headas-base-release> identifies the release of headas to be used as the foundation for the Astro-H release, and <release-candidate-baseline-identifier> names the Release Candidate

Baseline for the release. If the tests do not pass, no tag shall be applied. Instead, developers shall correct the problems and update the astrohdev baseline. When all updates have been made, the integrator shall again attempt to establish an Integration Baseline. The cycle shall repeat until the Release Candidate Baseline tag is successfully applied.

5.4 Release Baseline

Following additional independent verification and validation by the scientific staff, and sign-off by the Lead Developer and Customer, an approved Release Candidate shall be promoted to the status of ***Release Baseline***. The naming convention for Release Baselines is yet to be determined.

5.5 Creating Baselines

Whenever code meets the criteria for one of the above baseline definitions, the developer is responsible for tagging the code using the baseline name as the tag. For example, to add or promote code to the astrohdev baseline, one would go to the directory containing the code and type:

```
cvs tag -F astrohdev
```

Note that the -F flag will move the tag if the tag already exists, so this flag should only be used for baselines like astrohdev whose content is subject to change.

6 Testing Standards

The following discusses specific aspects of AstroH software testing standards, focusing primarily on unit tests. Discussion of so-called “thread tests” will be elaborated upon in future versions of this document (future build releases).

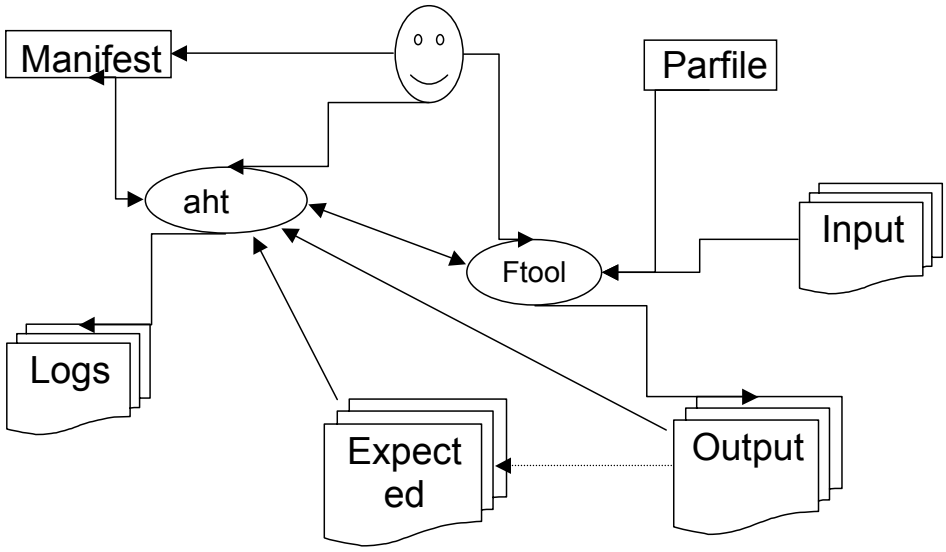
6.1 Compiled Unit Tests

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.

6.2 Application Unit Tests

The result of the software build installation is a set of UNIX/Linux command-line applications, which may be invoked directly 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.

The context diagram below provides an overview of the unit test scenario:



In the sub-sections that follow, the activities described by the diagram above are described in detail.

6.3 Initialization of the Unit Test Manifest, Directories, and Runtime

The Perl script 'aht' installed by the hmake provides a unit-test "harness". It can be used, in theory, at any point in the software development cycle. Assume one creates a new empty directory under ones' $HOME and runs aht the very first time from within that directory:

```
# start unit tests somewhere:
mkdir $HOME/utest ; pushd $HOME/utest
#
```

then:

```
# initialize the unit test directory structure:
aht -i
#
# or
aht –init
```

The above 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):

```
# ls -al ./ ./log ./etc yields:
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
```

6.4 The Unit Test Manifest (./aht_manifest)

Please note the directory listing above shows the file 'aht_manifest.pl'. This Perl script 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'. The 'ahdemo' FITS tool application (Ftool) is described in the next chapter.

An annotated example Aht manfest is provided in appendix M.

6.5 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 administrator may be needed to enable non-zero core dumps.

6.6 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.

6.7 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).

6.8 Aht 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 (Ftool developer or tester) 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):

```
# manual invocation of Ftool:
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:

```
# use aht Perl script to initializE and/or update the unit test area and perform tests:
aht -i
# or
aht -u
# 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). The aht Perl script writes its own log files under the "./log" sub-directory, while it redirects the Ftool app's. "stderr" and "stdout" into log files under the "./output" sub-directory. These log files should be inspected to determine the success or failure outcome of the unit test.

```
./:
.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

6.9 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.

6.10 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.

7 Libraries

James 4/27/12 10:31 AM

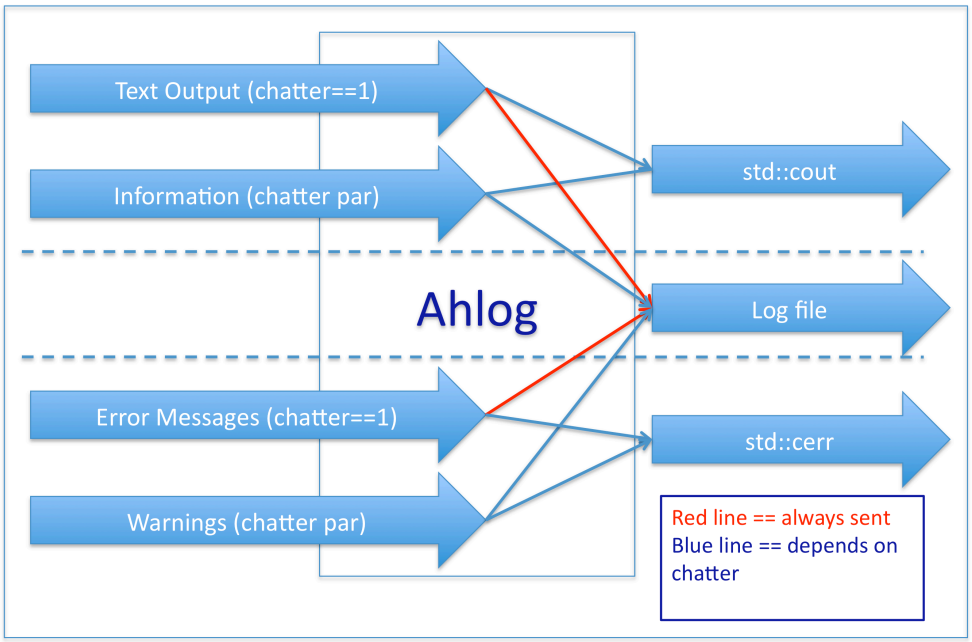
Comment: Mike to update to describe macros and current ahlog capabilities.

7.1 Ahlog: logging/output facility

The ahlog library contains logging and output facilities, which allow text output, warnings, informational messages and errors to be directed to the screen, standard streams, and/or to files. How much output actually appears is controlled by the tool's chatter parameter. An exception is that application-critical output and error messages are always written in the log file. As shown in Figure 3, client code can send textual messages to four distinct streams for the following purposes:

- **Ahlog::out():** used for textual output that is considered to be an end result of the algorithms performed by the code. This output is just as much a form of output data as data that is stored in FITS tables, and it is always written in the log file regardless of chatter level.
- **Ahlog::info(chatter):** used for textual output that is informational in nature. This output is not considered to be an end result of the tool, but is potentially useful information for the user.
- **Ahlog::err():** used to indicate tool malfunctions, or incorrect function or output. Output to this stream indicates that the software has encountered a condition that will prevent successful completion of the current task. Text directed to this stream is always written in the log file regardless of chatter level.
- **Ahlog::warn(chatter):** used for warnings of questionable tool behavior. Typically these would be caused by irregularities in data or input parameters. Messages to this stream are intended to warn the user of *possible* errors, but do not indicate a condition that would prevent the tool from completing execution.

Figure 3: Data Flow in ahlog



8 Build 0 Implementation

James 4/27/12 10:47 AM

Comment: In chapter 7:
7.2 David to add ahfits description.
7.3 James to describe ahgen.
7.4 Mike to describe ahtime.
New chapter: task description (James to write)
New chapter: requirements for task/algorithm specification (Lorella to write, input requirements)
New chapter(s) about Perl standards (libraries, source code standards) (team discussion needed)
New chapter for pipeline (Ron to write)

8.1 Libraries

The following libraries were implemented in Build 0:

- **libahlog:** Build 0 version was very primitive, just redirected everything to standard streams.
- Etc.

9 Appendix M – Sample Aht Manifest.pl file

```
#!/usr/bin/env perl
#
# declare and define/init test-tool hashes (of hashes):
# as our runtime manifest -- perl statements to be eval'd...
#
########################################## start of astro-h unit test manifest
###################################
#
# please do NOT change this hash variable name!
```

```
#
our %runtime_manifest = (
#
# the manifest provides direction to aht.pl regarding specifics
# of input and output files, including logs. the testheader should indicate something
# about the nature of the test and who when where it ius/was conducted.
# typically expect the testheader to be set at aht runtime via "aht.pl -i or -u"
# (or "aht.pl --init or --update"). but the developer can override updates by manually
# editing and setting the final attibute to "true" -- which indicates update
# should not reset this manifest entry:
testheader => {
  final => "false",
  describe => "testheader: indicates version, who, where, when",
  default => "version, accnt., hostname, date-time",
  valvec => ["v000", "me", "hostname", "today-now"]
},
#
# support either (t)csh or (ba)sh login/runtime...
# the indicated file should establish all required tool runtime
# environment variables (PATH, LD_LIBRARY_PATH, PERL* etc.)
setup => {
  final => "true",
  describe => "setup: runtime env. in (t)csh or bash",
  default => "./etc/setup.sh",
  valvec => ["./etc/setup.csh", "./etc/setup.sh"]
},
#
# aht.pl must be informed of the specific tool apps this manifest supports
# invoking "aht.pl -t" or "aht.pl -toolname" with the name of an existing
# binary app. will run the app. as well as updating this manifest entry with
# a new "default" app. path-name. i.e. "apt.pl -t foo" should indicate the
# foo binary currently residing in ./bin/.
toolname => {
  final => "false",
  describe => "toolname: binary executable to run/test",
  default => "ahdemo",
  valvec => ["ahdemo", "./bin/status", "./bin/fitsverify", "./bin/suzakuversion", "./bin/hxdpi",
"./bin/xispi", "./bin/xiscoord", "./bin/xissim"]
},
#
# aht.pl should pass along any cmd-line options it does not recognize as its own
# to the tool binary app. it executes/tests. if the tool app. is happy with
# the provided cmd-line options, aht.pl may update this manifest entry (maybe?)
toolcmdopts => {
  final => "false",
```

```
  describe => "toolcmdopts: optional command-line options to feed binary executable",
  default => "mode=h clobber=yes",
  valvec => ["mode=h", "clobber=yes", "mode=h clobber=yes", "-l", "-q", "-e"]
},
#
# a non-zero timeout informs aht.pl that the tool app. should be expected to
# run to completion within some given time frame. aht.pl may update this manifest entry.
tooltimeout => {
  final => "false",
  describe => "tooltimeout: in seconds -- 0 indicates forever",
  default => "20",
  valvec => ["20", "10", "5", "2"]
},
#
# tool exit status codes
exitstatus => {
  final => "false",
  describe => "exitstatus: expected status codes for success and known error / failure modes.",
  default => "0",
  valvec => ["0", "1", "2", "3", "4", "5"]
},
#
# the full directory-path to the calibration database shall be explicitly indicated here
# or perhaps via an environment variable or some such combination?
caldb => {
  final => "false",
  describe => "caldb: path to the calibration database used by the fits apps. to be tested",
#  default => "${CALDB}",
  default => "./caldb",
  valvec => ["./caldb"]
#  valvec => ["./caldb", "${CALDB}", "${HEADAS}/caldb"]
},
#
# the update ("aht.pl -u") behavior triggers a directory scan of the input directory(ies)
# with the results placed into this entry -- the default will typicall be the first file found.
input => {
  final => "false",
  describe => "input: location and names of data and ancillary files",
#  default => "",
  default => "ref.tbz2",
#valvec => ["./input/event_file_3.fits[events]", "./input/event_file_2.fits[events]",
"./input/event_file_1.fits[events]", "./input/astroh-short.fits", "./input/astroh-long.fits"]
  valvec => ["ref.tbz2"]
},
#
```

```
# the update ("aht.pl -u") behavior triggers a directory scan of the output directory(ies)
# that results in a copy of all currently available files to the expected_output
# directory(ies), and an update/reset of this entry.
# note that the filenames will be prefixed automatically with
"./expected_output/hostname.domainname/"
expected_output => {
  final => "false",
  describe => "expected_output: files that the test shall compare to the runtime results/outputs for
validation",
  default => "",
  valvec => ["ahcxxdemo_out3.fits", "ahcxxdemo_out2.fits", "ahcxxdemo_out1.fits", "astroh-
short.fits.md5", "astroh-long.fits.md5"]
},
#
# post procesing validation should include searches for specific keywords in output log
# files (stdout and stderr). this manifest entry must be manually edited and is not affected
# by an update. consequently it is always final.
logkeywords => {
  final => "true",
  describe => "logkeywords: optional list of keyword to seach in logfiles.",
  default => "all",
  valvec => ["success", "* warning", "** error", "*** fatal"]
}
); # runtime_manifest hash
#
########################################## end of astro-h unit test manifest
####################################
#
# the aht.pl test driver supports its own cmd-line options
# as expressed in the following perl statment to be eval'd
# our aht cmd-line options:
our %aht_cmdopts = (
help => {
  describe => "help: -h or or [-]-help synopsis and related info.",
  default => "brief",
  valvec => ["brief", "verbose", "veryverbose"]
},
#
tooltest => {
  describe => "tooltest: -t or [-]-tool or [-]-toolname [ftool name] binary executable of test -- ftool
cmdline options",
  default => "false",
  valvec => ["test_tool_name", "run the test on the indicated tool"]
},
#
```

```
filesys => {
  describe => "filesys: -f filesystem info. used by init and for runtime test",
  default => "all",
#  valvec => [ "./log", "./etc", "./lib", "./bin", "./input", "./output", "./output/stderr",
"./output/stdout", "./expected_output", "./expected_output/stderr", "./expected_output/stdout"]
  valvec => [ "./log", "./etc", "./input", "./output", "./output/stderr", "./output/stdout",
"./expected_output", "./expected_output/stderr", "./expected_output/stdout"]
},
#
init => {
  describe => "init: -i or [-]-init [ftool name] initialize filesystem and create working manifest,
and implicit update if possible.",
  default => "testheader",
  valvec => ["teastheader", "init the test manifest of this tool"]
},
#
debug => {
  describe => "debug: -d or [-]-debug [ftool name] exec gdb debugger on current tool binary.",
  default => "false",
  valvec => ["false", "true"]
},
#
env => {
  describe => "env: -e or [-]-env print runtime environment of test.",
  default => "false",
  valvec => ["false", "true"]
},
#
noop => {
  describe => "noop: -n or [-]-noop [ftool name] print runtime execution info.",
  default => "false",
  valvec => ["false", "true"]
},
#
param => {
  describe => "param: -p or [-]-par [ftool name] print runtime tool parameter info.",
  default => "false",
  valvec => ["false", "true"]
},
#
update => {
  describe => "update: -u or [-]-update [ftool name] recursively copy latest -- good -- test results
from runtime ./output subdir to ./expected_output.",
# the update option has become complicated a bit by the requirement to segregate tests by
hostname and date-time.
```

```
# assume the most recent output is something like:
./hostname/YearMonDay/Hour.Min.Sec/output"
# and ./output is a symlink to it, and update should recursively copy all ./output to
./expected_output.
  default => "false",
  valvec => ["false", "true"]
},
#
verbose => {
  describe => "verbose: -v[v] or [-]-[very]verbose print verbose dscription of runtime test
activities.",
  default => "false",
  valvec => ["false", "true"]
},
#
manifest => {
  describe => "manifest: -m indicate manifest-file of test",
  default => "false",
  valvec => ["false", "true"]
}
); # aht_cmdopts hash
```