CDF/DOC/TRACKING/PUBLIC/6851
January 23, 2004 (Version 2.0)

Htwish — A Program for Automated Testing and Analysis of SVX4 Chips and Hybrids

John Freeman, Paul Lujan, Marc Weber

Lawrence Berkeley National Laboratory

Abstract

Htwish is a program designed to perform automated tests on SVX4 chips and hybrids. It runs a comprehensive set of tests on a hybrid, allowing the user to easily evaluate the hybrid's performance over a wide range of situations. It can then take the results of the tests and perform an analysis of them, determining whether the hybrid meets quality standards. Htwish is very simple to use, and can quickly and easily produce test results.

Contents

1 Introduction to `htwish` 3

2 Installation of `htwish` 3

2.1 Required software tools and languages 3
2.2 Required program libraries 4
2.3 Required hardware 5
2.4 Organization of the `htwish` files 5
2.5 Obtaining and installing the main program 7
2.6 Using wafer probing data with `htwish` 9

3 Running Tests 10

3.1 Running the tests automatically 10
3.2 Changing the parameters used in testing 11
3.3 Manually running the tests 11
3.4 A brief overview of the testing process 12

4 The `htwish` Analysis GUI 13

4.1 Introduction to analysis 13
4.2 Starting analysis mode 13
4.3 Loading tests and analyzing hybrids 13
4.4 Working with histograms 18
4.5 Creating and working with travelers 20

5 Descriptions of the Tests 20

5.1 Serial line test 21
5.2 Pipeline pedestal map 21
5.3 Pipeline gain map/pipeline gain 2 22
5.4 Pipeline delay 23
5.5 ADC limits 23
5.6 Dynamic common mode suppression (DCMS) 24
5.7 Sparsification and sparsification with neighbors 24
5.8 Risetime 25
5.9 Multihit 26
5.10 Linearity 27
5.11 Capnoise 27
5.12 Currents 29
5.13 Getmap (no charge, with charge, and dynamic) 29

6 Working with `htwish` Code **31**

6.1 Changing the code 31

6.2 Important code 32

6.3 Code structure 34

A Appendix **34**

A.1 Troubleshooting / comments 34

A.2 Switching between SVX4 and SVX4.2 chips 37

A.3 A note on notation 38

A.4 Error table 38

A.5 How the .pat / .ini files work 39

A.6 Using `htwish` with `htmon` 39

A.7 Igor Volobouev's documentation (summary) 39

A.8 Contact info 40

1 Introduction to `htwish`

`htwish` is a very useful SVX chip test program, originally written by LBNL's Igor Volobouev for the SVX3 chip, and now updated to test the SVX4 chip. It is designed to automatically run a comprehensive set of tests on a chip or hybrid, testing all aspects of its performance. Complete analysis of a hybrid can be performed in approximately 15 minutes.

`htwish`'s operation is divided into two parts: Its first task is to run a set of tests on a hybrid, producing a database file which can be saved as a record of the hybrid's performance. Its second task is to allow the user to open these database files using an interactive GUI, and analyze the results. When using this GUI, the user can create histogram plots of the raw database information, as well as higher-level plots which analyze this raw data. Additionally, the user can request that the program perform an evaluation of the hybrid based on its database info file. In this evaluation, perhaps `htwish`'s most important feature, the program takes the tests and uses them to evaluate different components of the hybrid (channels, cells, etc.), then combines these evaluations into an overall pass / fail evaluation of the entire hybrid. Just as the user can save the test info to a database file, the user also has the option of saving all plotted histograms and evaluations for future reference.

`htwish` can be used to test individual SVX4 chips, either still on the wafer or mounted onto SVX4 chip test boards, and it can be used to test SVX4 hybrids, both standard 4-chip hybrids and 2-chip Layer 0 (L0) hybrids.

2 Installation of `htwish`

Installing `htwish` will take a few steps. `htwish` runs off of UNIX systems with PCI bus architecture (although the bus can be changed to ISA without much difficulty), and is written in a number of different languages (C and Tcl being the most important). There are a number of libraries and tools you'll need for the program to work, and in this section, I'll describe them all, as well as point you to where you can obtain them. Furthermore, I'll describe the directories you'll need to set up and the machine-dependent alterations to the code you'll need to make. If you have any trouble installing the program, or notice that this section is incomplete, don't hesitate to contact me (contact info is in the Appendix).

2.1 Required software tools and languages

Before obtaining and compiling the `htwish` source code, you'll need to establish a software development environment in which the program can run. The following programs will be needed:

The GNU C compiler `gcc`:

Your system almost certainly has this already. If it doesn't, you can probably re-write the source code's Makefile to work with another compiler.

The GNU `make` utility:

Again, your system probably has this. If it doesn't, it has *some* `make` utility which you can use, with appropriate modifications, to set up the program build.

The Tcl language / the Tk Toolkit:

The front end of the `htwish` program is written in Tcl (Tool Command Language). It's an interpreted language (i.e. doesn't need a compiler) which can call C routines, and which has a very useful extension called the Tk Toolkit. The Tk Toolkit provides a simple way for the programmer to create X Windows-based GUI's. In order to obtain these, go to the Tcl Developer's XChange website at `www.tcltk.com`. Note that there will be several versions of both Tcl and Tk; `htwish` was written for Tcl 8.0.5 and Tk 8.0.5 . Additionally, you'll need some support software for Tcl and Tk — specifically, BLT 2.4 for Tcl (`www.tcltk.com/blt/`) and the Tix 8.1.3 extension to Tk (`tix.sourceforge.net`).

Perl:

Some of the auxiliary scripts are written in Perl. If this isn't installed on your system, you should probably install it from, say, `www.perl.com`.

2.2 Required program libraries

`htwish` utilizes the functionality of a number of software libraries. To take a glimpse of the libraries you need, examine `htwish`'s Makefile (assuming you've downloaded the `htwish` source code, as described later). For a more in-depth look, read the following descriptions:

CERN libraries:

For computational purposes, `htwish` makes use of the CERN libraries. In order to both read about and obtain these libraries, go to `wwwinfo.cern.ch/asd/cernlib/index.html`. For more specific information related to installing the CERN libraries, go to `wwwinfo.cern.ch/asd/cernlib/install/install.html`.

HistoScope libraries:

The histogramming done by the `htwish` analysis GUI is accomplished using the HistoScope set of routines. Although this is free software available from Fermilab, the HistoScope files have been extended by Igor Volobouev, the original author of `htwish`. You can obtain these modified files from the bottom of `www-cdf.lbl.gov/~igv/hs`.

BerkeleyDB libraries:

`htwish` uses the BerkeleyDB database librares to create the database files mentioned in the introduction to this manual. The version we use is fairly old — version 2.6.4 (whereas the current version is version 4.x). This can be obtained from `fcdfsgi2.fnal.gov`, the path being `/cdf/home/igv/Svx3d/db-2.6.4`. Make sure to remove the file `config.cache` before building the code, as `config.cache` is a file which contains host-specific configuration info.

SVX Chip testing libraries:

Fundamental to all LBNL's software which tests SVX chips is a set of libraries designed to construct patterns to send to the chip, as well as interface a computer to our testing hardware setup. For documentation on these libraries, take a look at Maurice Garcia-Sciveres and Dipanjan Ray's 1998 paper, "The PATT Chip Test Setup", located at `www-cdf.lbl.gov/svxii/Hardware/patt02.ps`. To obtain these libraries, simply contact us here at LBL and we'll send you an archive of all the necessary files. Note that if you use a different hardware test setup (for example, the PTS setup developed at UIUC), you'll need to rewrite some of the lower-level functions that `htwish` uses to interface with the hardware. More details on this can be found on the PTS home page at `www.hep.uiuc.edu/home/trj/pts/index.html`.

Power supply interface libraries:

If you wish to measure the currents of your DUT, you'll want both a GPIB connection between your computer and power supply, as well as LBNL's power supply interface libraries. As in the case of the SVX testing libraries, simply contact us if you want a copy.

2.3 Required hardware

You will, of course, need the appropriate hardware to communicate with the hybrid; if you're planning on using the LBNL PATT board/scrambler board setup, you will need these boards, as well as a PCI interface board in the computer to communicate with them. To test Layer 0 (L0) hybrids also requires an additional adapter board. Testing wafers will require an appropriate wafer probing station.

Some of the tests also require additional hardware. Many of the tests use charge injection, for which an external voltage (VCAL) is typically provided through a DAC in the computer. We use a CIO-DAC02/16 from Measurement Computing (`www.measurementcomputing.com`); other cards would require a Linux-compatible driver and probably some modifications to the I/O libraries. If you do not have such a DAC, you can use the internal VCAL connection of the chip instead by editing all of the .ini files in your PatternsV2 directory (see section 2.4 below) and changing the VCAL connection bit (bit 129) from 0 to 1. (Note that this feature is available only for SVX4.2 chips.) However, the internal VCAL only provides one voltage, so tests which rely on changing VCAL (gain, linearity, capnoise) will not function. Also, the current measurement test, as mentioned above, requires a GPIB connection to a HP E3631A (or similar) power supply. Finally, the capnoise test is designed for situations where some channels on your chip are attached to external capacitors; see the description of the capnoise test in section 5.11 for more details.

2.4 Organization of the `htwish` files

`htwish` is installed in a large number of different places. There are a total of ten directories which store data:

1. The code directory: This directory contains all of the Unix scripts, C files, Tcl files, and assorted other files necessary to build and run the main `htwish` program. (Overall about 10Mb in size.)

   Note: In general, you should run `htwish` and its assorted other utilities from this directory, since it's where they expect to be run. Attempting to run from other directories may cause errors.

2. The V2 pattern directory: This directory contains the initialization (.ini) and pattern (.pat) files which control the running of the SVX4.2 chips. (About 3Mb in total.)

3. The pattern directory: `htwish` also supports running with the older SVX4 chips, which use a 190-bit initialization bitstream instead of the newer 192-bit initialization bitstream. This directory is used to hold these older patterns. (Also about 3Mb.)

4. The dbsupport directory: This directory contains assorted files which support the Berkeley DB functions in `htwish`. (By itself, only about 1Mb. However, this directory should also have space for temporary files, which can occupy up to about 100Mb or so.)

5. The database directory: This directory contains the Berkeley DB files which are produced by `htwish`. (Typically about 9Mb per standard hybrid, 4.5Mb per L0 hybrid, and 2Mb per single chip.)

6. The traveler directory: The analysis features of `htwish` produce travelers, which are small documents providing a concise summary of a hybrid's performance. Saved analysis results and quality trees are also placed here. (Typically about 1Mb per hybrid, if everything mentioned above is placed here.) It's recommended that this directory be web-accessible, as `htwish` is capable of generating HTML travelers and quality trees.

7. The logfile directory: Log files produced in testing will be placed here. (A full log, with all informational messages, is about 700kb for a standard hybrid. A short log with just the pass/fail results is just a few lines.)

8. The wafer probing data directory: `htwish` can use data previously obtained from wafer probing in running its tests; specifically, if certain capacitors have already been determined to be bad during wafer probing, `htwish` can mask them out during further tests. This is where this data should be placed; see section 2.6 for more details on how to use this feature. (This is just a few small text files; less than 100kb.)

9. The flat ASCII directory: This directory is where the flat ASCII format files generated by the testing will be stored if flat ASCII is turned on. Flat ASCII files are used by some of the analysis tools developed at Fermilab, and can be imported using these tools into ROOT histograms. If you are not planning on using these features, then you won't need to use flat ASCII. (These can be quite sizeable; about 10Mb for a single standard hybrid.)

10. The summary file directory: This directory is where the summary files generated by the analysis will be stored if summary files are turned on. (These are not too big; also less than 100kb.)

The first four items on this list are provided in separate .tar files. The database, traveler, and log directories start out empty and will be populated as you use `htwish` (though you'll still need to create the empty directories yourself). The wafer probing data you'll have to generate yourself; see section 2.6 for details. The last two are optional, and you'll need them only if you plan to use those features; again, they will start out empty and be populated as tests are run.

You can, of course, make some of these directories the same, but each one tends to accumulate a large number of files.

Make sure that your permissions are set appropriately. At minimum, everyone who wishes to run htwish should have read+execute permissions to the code directory, read+write permissions to the dbsupport directory, and read permissions to the patterns and wafer probing directories; everyone using htwish for running tests should have read+write permissions to the database, log, and flat ASCII directories; and everyone using htwish for analyzing data should have read permissions to the database directory and read+write permissions to the traveler and summary file directories.

2.5 Obtaining and installing the main program

So your computing environment is ready to support `htwish`, and now it's time to install `htwish` itself!

First, you'll need to set up the directories. Create the directories as specified above in whatever location you desire. (Make sure that you'll have adequate space, however; as noted above, some of the directories can grow to quite large sizes.) Now, the first four should be populated by the contents of the appropriate archive file, `code.tar`, `patterns.tar`, `patternsV2.tar`, and `dbsupport.tar` respectively. Contact us to obtain these files. Each tar file should be extracted using the command `tar xvf` *filename*, which will unpack the archive into the appropriate files. If you plan to use wafer probing data with `htwish`, read section 2.6 below. The other directories can remain empty for now.

Now that you've set up all of these directories, you're going to need to actually tell the program where all of them are. Here's where you'll need to set everything:

1. `hybtest.h`: Change `DATABASE_HOME_DIR` to your dbsupport directory. Change `LOG_DIR` to your logfile directory. Change `FLAT_ASCII_DIR` to your flat ASCII directory. Change `SUMMARY_FILE_DIR` to your summary file directory. Set the two occurences of `PATTERN_DIRECTORY` to your two pattern directories (the first is the SVX4 pattern directory, and the second is the SVX4.2 pattern directory).
2. `run`: Change `$DBFILEDIR` to your database file directory. Change `$DBSUPPORTDIR` to your dbsupport directory. If you are not planning on using wafer probing data, set `$USEWAFERDATA` to 0. (This is purely cosmetic, however; all it does is suppress error messages if wafer data is not found. Thus, if you want to use wafer data for some hybrids and not others, you can leave this at 1 without any harm.)
3. `getchips.pl`: Change `$WAFERDATADIR` to your wafer probing data directory.
4. `new_traveler.pl`: Change `$traveler_dir` to your traveler directory, and also `$logdir` to your log file directory.
5. `new_html_traveler.pl`: Change `$traveler_dir` to your traveler directory.

6. testdisp_globals.tcl: Where db_homedir is set, change it to your dbsupport directory. Where traveler_dir is set, change it to your traveler directory.
7. DB_CONFIG (in the dbsupport directory): Change DB_DATA_DIR to your database file directory. Change DB_TMP_DIR to the Tmp subdirectory of your dbsupport directory.

Now, there are a few more minor changes which must be made to the source code.

1. Make sure that all standard Unix utilities are correct

   htwish uses the Unix shells sh and tcsh, which it expects to be in /bin/, Perl, which it expects to be in /usr/bin/perl, and a number of standard Unix utilities (mv, ls, and so forth; see the top of db_include.tcl for the full list of utilities and expected locations). If your computer is configured unusually, you'll need to fix this. You can change all of these by hand (use grep to find what you need to change), but it's much simpler to create a symbolic link from the expected location to the actual location. Just type
   ln -s *<full path of file on your computer>* *<full path that htwish expects>*

2. Put your computer on the list of htwish's allowed hosts

   htwish has a feature that will prevent it from running tests on machines it doesn't know. To change this list, change @ALLOWEDHOSTS in run and allowed_hosts in hybtest.tcl. Alternatively, you can disable this check by setting $CHECKHOSTS to 0 in run and check_hosts to 0 in hybtest.tcl.

   Similarly, it will also not display tests on machines it doesn't know. This list, similarly, can be changed by changing allowed_hosts in testdisplay.tcl, or disabled entirely by setting check_hosts to 0 in testdisplay.tcl.

3. Customize the Makefile's compile options and libraries

   There will be three things you'll need to do with the Makefile. The first is to supply the Makefile with the local directories in which it can find your copies of the aforementioned libraries. These are set in CERN_DIR (the CERN libraries), MGS_DIR (the SVX chip testing libraries), HISTO_DIR (the HistoScope libraries), DB_DIR (the Berkeley DB libraries), and GPIB_DIR (the GPIB interface libraries).

   Next, you'll need to update the dependencies. Type make Dependencies and the Makefile should generate a list of dependencies and place it in a file called "Dependencies". Take the contents of this file and replace the section of the Makefile containing the dependencies with it.

   Finally, you'll need to customize the compile line options the Makefile uses; these options are the following, and should be included or not at your discretion:

   SVX41 or SVX42: One of these mutually exclusive options must be set to determine which type of chip is being used. SVX41 refers to the first model of SVX4 or SVX4 v2; SVX42 refers to the SVX4.2 chips used in preproduction (SVX4.2A or SVX4.2B).

   Note that setting this option will ensure that testing works properly, but many of the analysis settings also need to be changed if you are switching between SVX4 and SVX4.2 chips. See section A.2 in the Appendix for more details.

   LBL_DAQ: When this is set, htwish will assume that the data it reads back will have been formatted by the standard LBNL PATT setup (leading and trailing junk bytes, priority passing bytes, etc.) Without this defined, `htwish` will assume a "flat" data format, such as we see with the PTA/PMC system with no superfluous bytes. This setting is crucial in order for data integrity checking to work correctly.

`INVERTED_BIT9`: This define is only used by the serial line test; it is needed if the priority out signal has its polarity flipped by the board the chip/hybrid is on.

Important note for Layer 0 hybrids: The L0 adapter board has the priority out signal flipped. You do **not** have to change this flag and recompile if all you're doing is testing a L0 hybrid. If `htest` is given a L0 hybrid, then it will automatically take care of this particular flip.

`FLAT_ASCII_SUPPORT`: If this is defined, certain tests will produce flat ASCII (CDF format) versions of their output data, in addition to the .db files. These flat ASCII files will be placed in the flat ASCII directory.

`SUMMARY_FILE_SUPPORT`: This feature allows certain tests (risetime, e.g.) to produce a formatted ASCII summary file of their results. These summary files will be placed in the summary file directory. Such files are used in root macros for analysis outside of `htwish`.

Note that while flat ASCII files are produced during testing, the summary files are not produced until the data is loaded into histograms in the analysis GUI, either by loading the individual tests in question or by running an analysis on the hybrid.

`OUTPUT_MODE`: If this is set, large amounts of ASCII data of the results read back from the chip/hybrid are dumped to the console. Useful for debugging.

`CAPNOISE_SUPPORT`: A feature, primarily written in order to help choose proper rise-times for the capnoise test, which causes the risetime test to loop over all 16 values of preamp speed (rather than only 3).

Make sure your program has an acceptable set of compile line flags! Botching this is a common source of program errors.

4. Compile the program

   Within the code directory, simply type "make", and the Makefile will do its job. Please note that a compiler will only re-compile a file that's been modified since the last compilation. A consequence of this is that, if you've changed some of the compile-line defines, you'll need to update all source files which recognize these defined quantities before compilation. A very simple way to do this is to type

   `grep -l` *compileflag* `*.c | xargs touch`

And that's how you install `htwish`....

2.6 Using wafer probing data with `htwish`

If a chip or chips in your hybrid has known bad capacitors, then this will cause some tests involving sparsification to fail, since they will be expecting that capacitor to be read out and it won't be. To prevent the accumulation of spurious errors, `htwish` has a feature by which known bad capacitors can be specified, and these capacitors automatically masked out in the testing. (They will still come up as bad in the appropriate parts of analysis, of course, but they won't cause more tests to fail.)

To automatically use these capabilities, you will need to create a wafer probing directory, as specified above. This wafer probing directory should contain the following:

1. A file called `hybridchips.txt`. This file should contain a list of the chips in each hybrid. Each line in this file should have the form
   `h060 19A1-338 19A1-339 19A1-340 22G7-225`
   where the first field is the name of the hybrid, followed by the numbers of each chip in that hybrid, in order. Each chip should be specified in the form wafer-chip number.
2. For each wafer, a file called *wafername*`_chips.txt` (that is, for wafer 19A1, the corresponding file would be `19A1_chips.txt`). Lines should look like this:
   ```
   41 B good
   42 A bad 10-39 16-25 2-36
   43 B fair 9-5
   ```
   where the first field is the chip number, the second field is the chip type (A or B), the third field is the overall quality of the chip (good, fair, or bad), and the last field is a list of bad capacitors. All bad capacitors should be listed in the form cell-channel. Note that for purposes of this list, cells should be numbered 0 to 45 (not 1 to 46, as the actual chip numbers them). This is because `htwish` reports cells as being from 0 to 45, so if `htwish` is used to find the bad capacitors, it's easier to use those numbers directly. Channels are from 0 to 127, as always.

   If the chip has other problems than bad capacitors, then these can also be placed in the file. Here's an example:
   `46 B bad channel 37`
   `htwish` does not know how to deal with these automatically, but it will report the problem to you when it is run.

Also, after placing all of the wafer data files in your wafer probing directory, you will need to edit the program `getchips.pl` and change `@WAFERS` to a list of your wafers.

You can also manually give bad capacitor data to `htwish` when you run it; see section 3.3 for details.

3 Running Tests

3.1 Running the tests automatically

The simplest way to run tests in `htwish` is to use the automatic run script, `run`. All you have to do is type `run`. The script will prompt you for the name of the hybrid. If wafer probing is enabled, the script will then find any bad capacitors and pass them on to the program. It will also check to make sure you're in the right directory and make sure tests have not already been run for this hybrid (if they have, it will give you the opportunity to delete the old tests). It will also create a rudimentary lock to prevent other occurrences of `htwish` from attempting to access the database file in progress. (This is a frequent cause of crashes in older versions of `htwish`.) It will then appropriately invoke `htwish` to run the tests.

While the tests are running, various status messages will be displayed to the console. At the end of the tests, a list of whether the hybrid passed or failed each test will be displayed.

IMPORTANT NOTE: The pass/fail result that is displayed for a given test is *not* a complete picture of whether the hybrid passed that test. All a "pass" result means is that all of the expected channels read out without any readback error during that test. (Conversely, a "fail" result means that there were readback errors in that test.) It does *not* actually have any analysis of the channel data. In order to analyze the data read back from the hybrid to see if it was meaningful, you must run an analysis on the data file obtained during testing.

3.2 Changing the parameters used in testing

The run script passes four parameters to `htwish`. These are as follows:

1. *hybname*: `run` gets this from user input. This name is the name given to the hybrid, and the .db file will have this name as well.
2. *nchips*: This is the number of chips on the hybrid. If wafer-probing data is being used, then `run` will get this number automatically from the data. If not, it will assume 4 chips unless the hybrid name contains "L0", in which case it will assume a L0 hybrid with 2 chips. To change this, change `$DEFAULTNUMCHIPS` or `$DEFAULTL0CHIPS` in `run`.
3. *hyb_type*: This is the type of the hybrid. Currently "standard" or "L0" types are supported. Again, by default, `run` will assume a standard hybrid unless 2 chips are specified, in which case it will assume a L0 hybrid. If you want to change the default type, change `$HYBTYPE` in `run`. (If, for some reason, you don't want 2-chip hybrids to be assumed to be L0 hybrids, then you also should change `$DEFAULTL0CHIPS` to some number other than 2.)
4. *testtype*: `htwish` supports three different types of tests. "test1" tests are designed to be run before burn-in, and "test2" tests are designed to be run after burn-in. Only one of each may be performed on a single hybrid. You may perform an unlimited number of "user" tests at any time. `run` assumes that you are performing a "test1" test; if you want to change this, then change `$TESTTYPE` in `run`.

`run` also automatically passes a list of bad capacitors to `htwish`. If you wish to put these in manually, the simplest way is to create a set of wafer-probing data for the chips in question following the directions in section 2.6. You can also just run `htwish` manually following the directions below.

3.3 Manually running the tests

You may also run `htwish` without using the `run` script with the following command:
`htest.tcl` *hybname hyb_type nchips testtype badcaps*

Here, *hybname*, *hyb_type*, *nchips*, and *testtype* are the four parameters described above. *badcaps* is an optional list of bad capacitors. These should be in the format cell-channel, where cell runs from 1 to 46 (**not** 0 to 45 as the wafer-probing data does) and channel runs from 0 to the maximum number of channels on the hybrid (511 for a 4-chip hybrid, for example). If you have multiple bad capacitors, they should be separated by spaces.

3.4 A brief overview of the testing process

Depending on how you count them, there are about fifteen different tests which can be run on the hybrid during testing. Each examines a different feature of the chip: pedestal, gain, sparsification, the pipeline pointer, and so forth.

Some of these tests exist only to make sure that the hybrid is functioning properly, and so the pass/fail provided at the end of the test run is all you need (for example, the serial line test). However, most of these tests produce data which is designed to then be further analyzed in the `htwish` analysis GUI (for example, the pipeline pedestal map test).

A test typically runs in the following fashion: it initializes the chip or hybrid using an .ini file from the appropriate patterns directory. It then runs at least one pattern on the chip, where a pattern is defined by a .pat file. After running the patterns, the program will read back the data which the hybrid clocks out, and then analyze its integrity. This process is usually repeated numerous times in a single test, with the initialization stream often being altered by a loop over one or two of its variables.

Essentially all tests contain a data integrity check; for those pass/fail result tests mentioned earlier, the result of this check is all that matters. The check works as follows: every time an event is read back, a function (`hyb_parse_data` — see section 6 of this Note) checks the data stream and makes sure that it finds the data formatted correctly. That is, it makes sure it receives the chip ID, cell ID, and channel IDs that it expects.

In case an error occurs and a nonzero value is returned, an error table will be created and written to the .db file. During the `htwish` GUI analysis, it will be possible to look at a 2D histogram which shows the number of events for every chip on the hybrid which failed, and what the error codes were. (See section A.4 in the Appendix for a full listing of errors.) Error values which apply not to a specific chip but rather to the hybrid as a whole (such as 0 or 1) are automatically assigned to the "last chip" in the 2D histogram — actually taken to be one chip beyond the maximum chips `htwish` allows on a hybrid. Those tests which are merely exposed to a data integrity check will write relatively little data to the .db file; either an error table in case an error occured, or just a simple "pass" integer if the test was passed. More interesting are those tests for which the data itself is of interest. In this case, when `htwish` is run in its analysis GUI mode, the user will be able to look at the raw data in histograms, as well as be able to produce analysis histograms calculated from this raw data. When this type of test is used in the pass/fail of a hybrid, it is considered a success not simply because it passed the integrity check, but also because its results fell within expected bounds.

4 The `htwish` Analysis GUI

4.1 Introduction to analysis

Getting `htwish` to run the tests on the hybrid and produce a .db file is an utterly trivial matter: just press enter, and then take a trip to the snack machine. On the other hand, interacting with the program becomes interesting when `htwish` is run in its GUI analysis mode. Here, as mentioned earlier, you can do a number of things pertaining to the analysis of the data you collected in the testing phase. You can get the program to histogram the raw data from a .db file, produce analysis histograms for this data, and save these histograms to files which can be looked at in another `htwish` GUI session. You can also get the program to take the results from a .db file, and use a hierarchical method to pass or fail the test hybrid, displaying a nice "quality tree" as a means of visualizing the results (more on this later in this section).

4.2 Starting analysis mode

Opening up `htwish` in its analysis mode is quite simple: just type

```
htwish testdisplay.tcl
```

This will pass the `testdisplay.tcl` Tcl script to the `htwish` executable, a script which `htwish` uses to begin its analysis mode. Two windows will pop up: the main `htwish` analysis window (fig. 1), and a HistoScope window (fig. 2). The latter is used to display HistoScope histograms, so don't close it (otherwise, you'll lose the ability to display histograms), but you shouldn't need to interact with it directly (you can use this window to perform specific HistoScope operations on individual histograms, but this is almost never necessary). When these windows are opened, the console window will no longer accept input, but will display various status and output messages.

After opening the main windows, `htwish` will take a few seconds to load in the list of hybrids and tests. When it is done, it will display the list of hybrids in the upper-left window and the list of tests in the upper-right window. The histograms that are produced will be displayed in the lower pane (as illustrated in figure 1), although this begins empty.

4.3 Loading tests and analyzing hybrids

Here's a description of each of the top set of buttons in the GUI, which are used for loading tests and analyzing hybrids. The bottom set, used for working with individual histograms, is described in the next section.

Load tests

It's often desirable to visually inspect the raw data of a chip or hybrid. To do this, highlight the hybrid (or hybrids) whose data you want to look at, and then highlight all the tests whose results you're interested in. (Use control-clicking if you wish to select multiple, non-contiguous items, from either list.) Then hit this button. This will load the raw data from the db file and

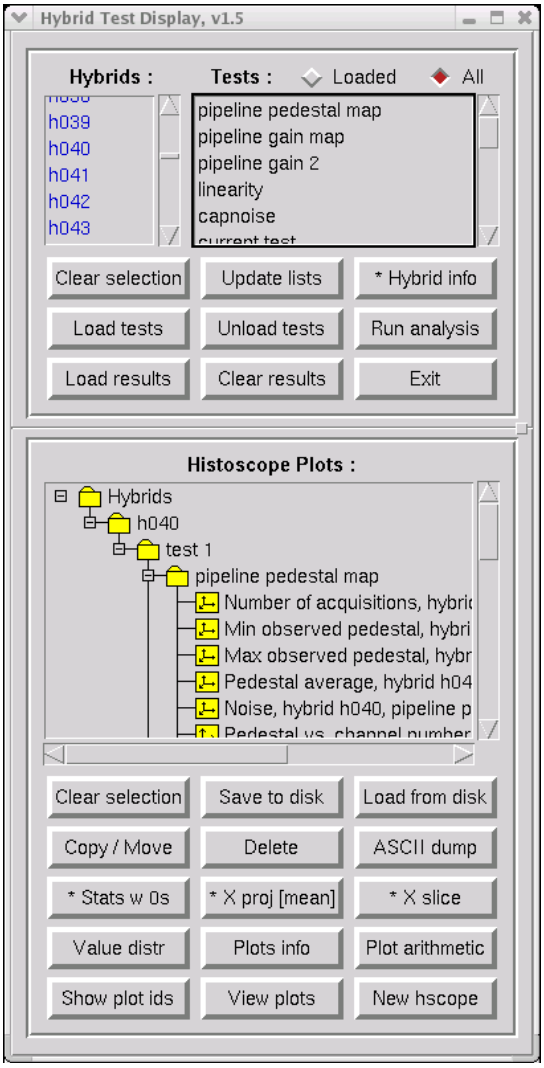

Figure 1: The test display window.

place it into a number of histograms. (For some tests, such as risetime, the histogramming also will involve some curve-fitting, which can be time-consuming.) The results of this loading will be displayed in the console window.

When the loading is complete, a tree of the histograms you loaded will appear in the lower window. To view one of these histograms, simply double-click on it. See figures 3 and 4 for examples. To perform more sophisticated operations on the histograms, see section 4.4 below.

Unload tests

This button will remove the histograms for the selected tests from the histogram list. Usually not necessary unless you want to keep things really neat and clean.

Clear selection

This button will unselect all selected hybrids and tests.

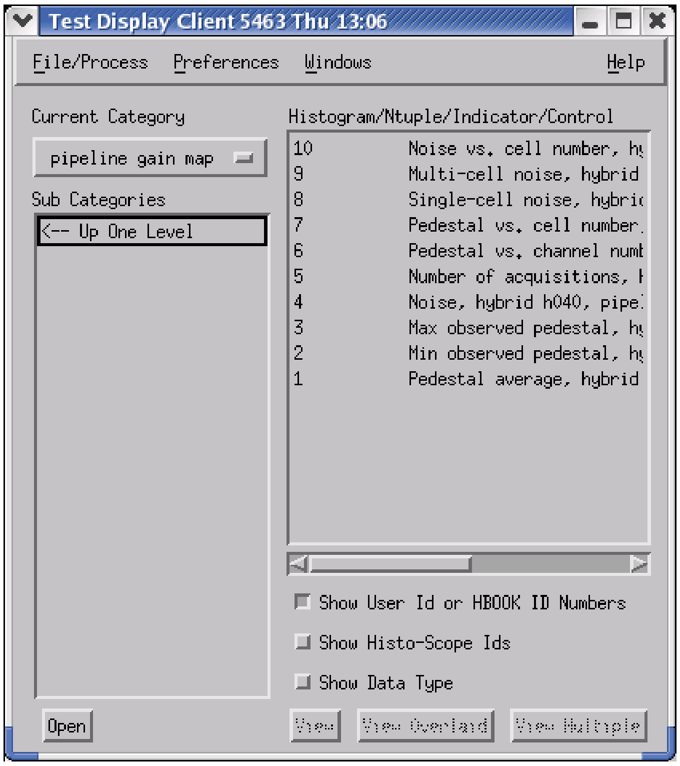

Figure 2: The HistoScope window, showing a sample list of histograms.

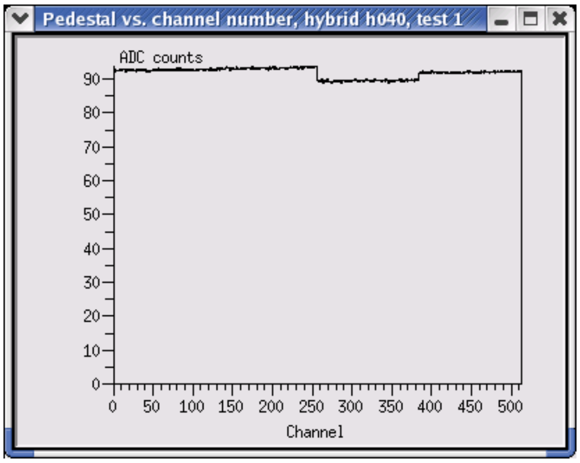

Figure 3: A sample 1-dimensional histogram, showing pedestal vs. channel. The different levels are due to the differences between individual chips in the hybrid.

Update lists

This button will reload the list of available hybrids and tests from disk. It will take a few seconds to complete.

Hybrid info

This button will display some very basic information about the selected hybrid or hybrids.

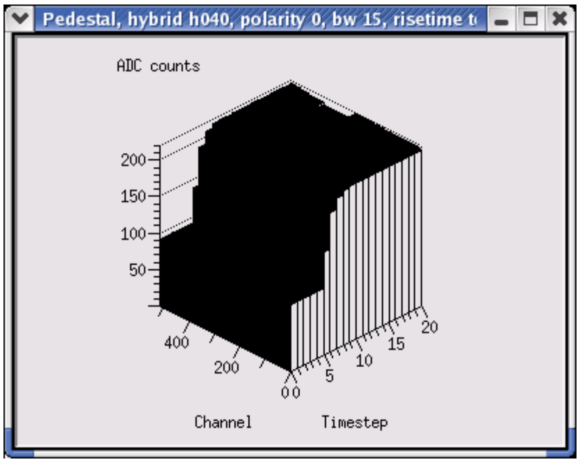

Figure 4: A sample 2-dimensional histogram. This histogram is used for the risetime test; it shows the measured ADC counts vs. time and channel.

Run analysis

This button is, in many ways, the heart of `htwish`. When you use it on a selected hybrid (or hybrids), it will take that hybrid, load in all of the tests, and perform a series of cuts on the results of these tests. Using all of these results, it will then make a final judgment of the hybrid's quality based on its performance in the cuts.

Pressing this button will analyze the hybrid, a process which may take a few minutes. At the end of the analysis, a quality tree representing the final quality judgment of the hybrid will be displayed. See figure 5 for an example.

Analyzing a hybrid will produce two more results: histograms for inclusion in a traveler will be produced and placed in the traveler directory (in a subdirectory with the name of the hybrid), along with a brief summary of the analysis results. These histograms will also be displayed in the histogram window in the "Travelers" section.

Secondly, a series of histograms representing data for all hybrids analyzed in this session will be updated (or created, if this is the first hybrid you've analyzed in this session). These histograms, which are displayed in the "test stats" section of the histogram window, represent the aggregate data for all hybrids analyzed. For example, the "pedestal distribution" histogram will have one entry for each chip, with the pedestal for the chip. You can use this histogram if you've analyzed a large number of chips to look at the overall pedestal distribution of the set, for example.

The quality tree is fairly self-explanatory: the hybrid is broken down into components — as a whole, as chips, cells, channels, all the way down to pipeline capacitors (one for every cell/channel combination). Every test run on the hybrid contains one or more aspects which factor into the evaluation of one or more components on the hybrid. Some of these tests, as described earlier in the manual, simply provide pass/fail info to the program; others provide histograms of their results, which `htwish` can use cuts on as a means of analysis. For every hybrid component, for every test aspect, there's a colored circle attached next that component / test aspect's name to describe `htwish`'s findings. The color key is found at the bottom; just

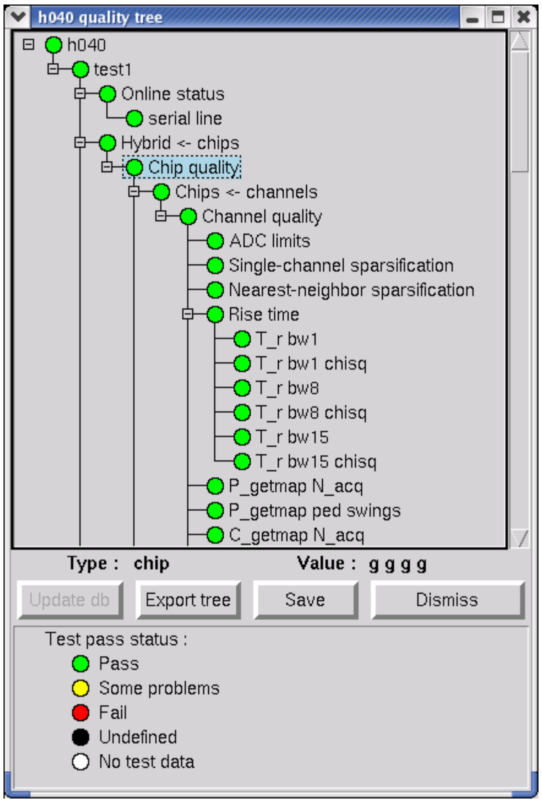

Figure 5: A sample quality tree.

briefly, green is pass, yellow is fair (which, under current cuts, means some bad capacitors but not enough to fail the hybrid), red is fail, white means that the test was not run, and black is undefined. Note that quality judgments propagate up the tree — for example, the hybrid is green if and only if all of its chips are green; if one or more of its chips are yellow, the hybrid will be yellow, and if one or more of its chips are red, the hybrid will be red. (A hybrid can also be red if it fails one of the hybrid-level tests — for example, current — even if all of its individual chip tests are green.)

Clicking on a given item in the quality tree will display, at the bottom of the window, the "type" of the test (i.e. whether it applies to hybrids, chips, channels, cells, or capacitors), and its results, as illustrated in figure 5. For some tests, the result will simply be displayed in a word or letters (g for good, f for fair, b for bad); if a number is displayed, that means you can look at a histogram of the results of that cut by double-clicking on that item. In these histograms, a 1 indicates a channel, cell, or capacitor that failed the cut in question. So, for example, figure 6 shows that the capacitor in cell 40, channel 212 failed the "pedestal residuals" cut.

At the bottom of the quality tree window are four buttons. "Update db" is not currently supported. "Export tree" will save an HTML-format copy of the tree in the traveler directory. "Save" will save the full analysis results in the traveler directory. "Dismiss", as you might expect, will dismiss the window.

Note that if you attempt to analyze a hybrid on which the full set of tests has not been run, `htwish` will produce an error and need to be restarted. To change the list of tests used in analysis, see section 6.

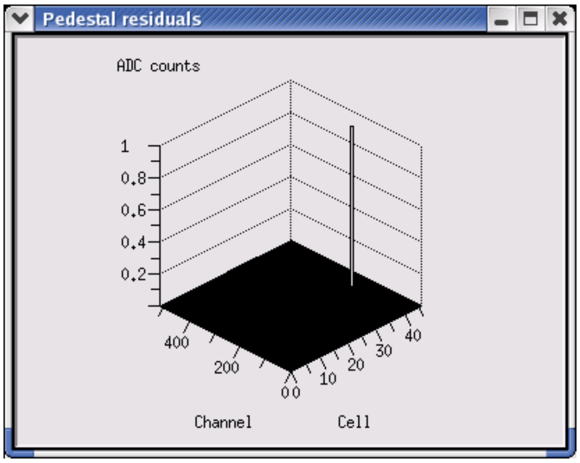

Figure 6: A sample cut results histogram. 0 indicates that the capacitor (or cell, or channel, etc.) passed the cut; 1 indicates failure.

Often, you'll want to analyze hundreds of chips during one `htwish` run; in this situation, it may become a little awkward to have your monitor display hundreds of quality tree windows. Changing the `htwish` code to suppress this feature is extremely simple; refer to section 6 to find out how.

If you press the "Run analysis" button on a hybrid which has already been analyzed, then the program will save you time and just redisplay the quality tree, without redoing the whole analysis.

Load results

This button will load analysis results that you saved from the quality tree window (see above). Note that loading analysis results in this way will not update the aggregate "test stats" histograms; you'll need to save those individually.

Clear results

As noted earlier, pressing "Run analysis" on a hybrid which has already been analyzed will redisplay the quality tree, not actually re-run the whole analysis. If you really do want to reanalyze the hybrid, then use this button to clear the analysis results before proceeding.

Exit

This button should be pretty self-explanatory.

4.4 Working with histograms

As mentioned earlier, to display a histogram in the histogram pane, simply double-click on it. These histograms have been implemented with the HistoScope package, which provides much useful functionality for the user. Two features especially deserve mention. If you right click while the mouse pointer is over the histogram, a panel of options will appear. One of the most useful options is called "Statistics"; this will permit you to see data like the number of entries, average, etc. in your histogram. Also, if you point to an axis and drag in one direction, you can expand or contract the scale of the axis and consequently the range of data you can see. This is useful for examining local details in a histogram. 2D histograms can be rotated simply by clicking on the histogram and dragging in the desired direction.

Here's a description of the buttons in the lower half of the test display window, used for working with histograms.

Clear selection

Like its twin in the top half, this will deselect any selected histograms in the histogram display window.

Save to disk, Load from disk

If you want to save a histogram or histograms to disk so that you can easily reload it later without having to redo the whole process that produced it (particularly convenient for the aggregate statistics), then use these buttons. These files should typically be saved with a .hs extension so that HistoScope can recognize them.

Copy/Move, Delete

These buttons allows you to copy a histogram, move it to a different category, or remove it entirely. Not particularly useful.

ASCII dump

This button will write out an ASCII dump of the selected histogram or histograms to a selected file. This can be quite useful if you want to get the precise values of the bin contents in the histogram.

Stats

This button will display a number of statistics (mean, median, standard deviation, etc.) for the selected histogram to the console window.

X projection (mean)

This button will take a 2-dimensional histogram and project it onto the X dimension, using the mean value along the Y dimension.

X slice

This button will take a 2-dimensional histogram and display a 1-dimensional slice for a given value of the X coordinate. You can adjust this value using a provided slider.

Value distribution

This button will create a new histogram with the value distribution (that is, the frequency that each given value appears) of the selected histograms.

Plots info

This button will display some basic bookkeeping information about the selected histogram.

Plot arithmetic

This button will allow you to combine two separate histograms via addition, subtraction, multiplication, or division to create a new histogram.

Show/hide plot ids

This button will display the HistoScope IDs of each histogram (or hide them, if they're already being displayed).

View plots

This button will display the selected plots. For a single plot, it's the same result as just double-clicking it.

New hscope

This button will bring up a new HistoScope window. Useful if you've accidentally closed yours.

4.5 Creating and working with travelers

`htwish` has the capability to create "travelers", which are documents containing a brief summary of a hybrid's performance and results. As mentioned previously in this section, upon running an analysis, data suitable for travelers is saved to the traveler directory. In order to turn this into a full-fledged traveler, you must run `new_traveler.pl`. To do this, merely type `new_traveler.pl` *hybridname* from the code directory.

This will open an Emacs window allowing you to edit any information about shorts and opens, and any comments on the hybrid. When you're done with this editing, quit the Emacs session. The script will then convert the traveler histogram data to PostScript graphs, and produce the traveler. It will then display the resulting document on screen, and (if you approve of the results) create a PostScript version of the traveler which you may then print out.

After running `new_traveler.pl`, you may then create an HTML-formatted version of the traveler by running `new_html_traveler.pl`. This will take the PostScript graphs and convert them to PNG, and produce an HTML version of the traveler. Just specify the hybrid names you want to generate an HTML version for; `new_html_traveler.pl` can take multiple hybrid names as arguments if you want to do a large batch at once.

5 Descriptions of the Tests

This section describes the individual tests. Each test description is divided into a number of parts: first, a description of the test; second, a description of the histograms produced; and third, a description of how that test is used in `htwish`'s analysis. For some tests which have common problems, a description of these problems and how to solve them is also provided.

5.1 Serial line test

- *test description:* The serial line test is used to make sure that data can get properly clocked in on the priority in line, through the shift register and read out on the chips' priority out line. The test proceeds as follows: first, enough 0s are clocked in to the hybrid that all chips should be filled with 0s. Then a chip's worth of 1s is clocked in, and data is read out from the hybrid. What should happen is that the 1s "push out" the 0s, so we expect to read out a chips' worth of 0s. This process is continued until now the shift registers on the hybrid are filled with 1s; then the reverse process occurs. Every step along the way, the test makes sure that the data it reads out is either a pure chips' worth of 1s or 0s. Also, the test makes sure that it gets the expected number of chips' worth of 0s and 1s. Here at LBL, we've noticed that under some hardware setups the bits that get read back are the inverse of what's expected. In this case, `htwish` should be compiled with the `INVERTED_BIT9` option set in the makefile.

  The test will pass if it reads out the 0s and 1s expected, and fail otherwise. If the serial line test fails, `htwish` will immediately stop testing, since if the hybrid can't be properly initialized, then the other tests can't be performed anyway.
- *histograms:* None. The serial line test is pass/fail only.
- *analysis:* The pass/fail result is applied to the hybrid in the "serial line" cut.
- *common problems:* As mentioned above, the most common problem is having a board on which priority out and its inverse are flipped, as is the case here at LBL. If you're experiencing problems, then try adding the `INVERTED_BIT9` option to the Makefile (or if it's already there, try taking it out). Note that the L0 adapter board also flips these signals, but `htwish` automatically allows for this if given a L0 hybrid.

  A much more subtle problem with the L0 adapter board is that (unless you're testing a full stave) you need to make sure that the jumpers on the board are set to the "phi" position rather than the "phi+z" position. Otherwise, the priority out signals are not returned from the hybrid.

5.2 Pipeline pedestal map

- *test description:* This test will examine the non-charge-injected readout value of all the pipeline capacitors on the hybrid. It does this looping over all 46 cells on the hybrid's chips; for each cell, an event is read out. By reading out an event, all the channels on the hybrid for that given cell will be read out.

  The test uses a cycling pattern so that each new iteration of the pattern will use a different pipeline cell; each individual cell is run 20 times.
- *histograms:*
  - Number of acquisitions: a 2D histogram showing the number of times each cell and channel was read out. Should be 20 for all.
  - Min/max observed pedestal, average pedestal: 2D histograms showing the mininmum, average, and maximum pedestal value read out over the 20 iterations.
  - Noise: a 2D histogram showing the RMS noise for each capacitor over the 20 iterations.

- Pedestal vs. channel number: a 1D histogram showing the pedestal averaged over all cells for each channel.
- Pedestal vs. cell number: a 1D histogram showing the pedestal averaged over all channels for each cell.
- Single-cell/multi-cell noise: 1D histograms showing the noise vs. channel, calculated two different ways. In single-cell noise, only the results from a single cell are calculated in determining noise. In multi-cell noise, the noise is calculated over all cells.
- Noise vs. cell number: a 1D histogram showing the noise calculated over all channels for a given cell.

• *analysis:*
  - First, the overall median pedestal for each chip is calculated, and checked to make sure it is within the cut boundaries. This is the "Pedestal" chip-level cut.
  - Then, the pedestal level for each individual capacitor is checked to make sure it is within the acceptable range of the median pedestal. This is the "Pedestal residuals" cap-level cut.
  - Finally, the noise level for each individual capacitor is checked to see if it is below the acceptable level. This is the "Pedestal noise" cap-level cut.
  - Running an analysis also adds the chip pedestal levels, the pedestal residuals, and the noise distribution to cumulative histograms.

• *common problems:* Typically, pipeline pedestal map will not fail when running tests unless the chip has serious problems (or there are connection problems), since it will fail only if there are readback errors. During analysis, it is exceedingly rare to fail the "pedestal" cut; usually, this only happens when the chip is stuck at 255, which is usually indicative of a serious chip internal error. If capacitors fail the "pedestal residuals" cut, then they are probably stuck at a certain value and are genuinely bad. The "pedestal noise" cut is more difficult to judge — sometimes, it's indicative of a real problem with the chip, and sometimes it seems to be a peculiarity of the testing setup. You'll have to judge.

5.3 Pipeline gain map/pipeline gain 2

• *test description:* These pair of tests are essentially the same as pipeline pedestal map, except that they have charge injection on. The "pipeline gain map" applies the standard voltage (currently 0.5 V), while "pipeline gain 2" applies a somewhat smaller voltage (currently 0.3 V). During analysis, the gain is then calculated from these two data points.

  Note that proper calculation of the gain requires correct calibration between the voltage sent by the computer DAC and the voltage received by the chip or hybrid; see section 6 for more details.

• *histograms:* Exactly the same set as for the "pipeline pedestal map" test, except of course the pedestal values will be higher.

• *analysis:* Very similar to the analysis for the "pipeline pedestal map". First, the median gain for each chip is calculated and checked (the "gain" chip-level cut); then, each capacitor is checked to make sure it's within an acceptable range of the chip-wide gain (the "gain residuals" cap-level cut), and finally the noise for each capacitor during the gain test is checked (the "gain noise" cut). The chip gain, gain residuals, and gain noise are also placed in the cumulative histograms.

- *common problems:* Like "pipeline pedestal map", this test usually only fails during analysis. In our experience, about 99% of bad capacitors will show up here, since by far the most common failure mode for a capacitor is to be entirely dead (i.e. not respond to charge injection at all), which will cause it to fail the "gain residuals" cut. (Capacitors which only partially respond will also appear here.) The chip gain cut is not well determined — in our experience, if you have an otherwise-good chip whose gain is slightly outside the cut boundaries, feel free to change the boundaries. Be aware that the "gain noise" cut should be higher than the "pedestal noise" cut, since the noise tends to be higher with charge injection.

5.4 Pipeline delay

- *test description:* This is primarily a test of the pipeline cell pointer. As in the previous tests, every pipeline cell is cycled over. At each pipeline cell, the same pattern is run, but the trigger delay (latency) setting in the initialization stream is varied from 1 to 42. In this sense then, the pipeline cell pointer is tested 42 x 46 = 1932 times.

  This test is purely pass/fail: the only thing that matters is whether the data reads back successfully.
- *histograms:* None.
- *analysis:* The pass/fail result is applied to each chip in the "delay settings" cut.

5.5 ADC limits

- *test description:* The purpose of this test is to make sure the ADC latches are working correctly. The initialization parameters are set so that none of the channels are injected into, but due to a slow ramp and high pedestal value the ADC counter will always reach 255 — regardless of the pedestal voltage in the hybrid's channels. The channels on the hybrid will be read out for different chip initialization values of counter modulo. Since counter modulo will always be reached before the preamp fires, it's expected that the value read out for every channel will be the counter modulo value. Some of the modulo values are complementary in gray code, the mechanism by which the ADC latch stores the readout values. In this fashion, it is possible to determine whether or not bits are getting stuck in the ADC latches.

  This is also purely a pass/fail test, since `htwish` will do the comparison during the testing phase.
- *histograms:* None.
- *analysis:* The pass/fail result is applied to each channel in the "ADC limits" cut.

5.6 Dynamic common mode suppression (DCMS)

- *test description:* This is one of the more sophisticated tests performed by `htwish`. First, the chips on the hybrid are initialized with RTPS (real-time pedestal subtraction) on. The test starts with charge injection turned on for nearly all channels on the hybrid. In these circumstances, RTPS will not work properly: the chip will think that the injected channels are pedestal, and so the subtracted value will be the injection, not the pedestal value. Then, the number of injected channels is gradually decreased, until RTPS begins to function as designed, and the pedestal is properly subtracted off.
- *histograms:*
  - Suppression curve: a 2D histogram showing the average ADC counts for each chip as a function of the number of channels at pedestal.
  - Suppression curve stdev: a 2D histogram showing the standard deviation for the ADC counts on a given chip as a function of the number of channels at pedestal.
  - Location: a 1D histogram showing the location (in terms of number of channels at pedestal) at which the RTPS began to work properly for each chip.
  - Width: a 1D histogram showing the width (in terms of number of channels) of the above transition between RTPS not working and RTPS working.
  - Low level/high level: 1D histograms showing the low level and high level of ADC counts as determined by the curve fit.
  - Fit chisq: The location, width, and levels are determined by fitting a curve to the "suppression curve" data. These 1D histograms shows the goodness of this fit as a function of the chip.
- *analysis:* The location, width, and fit chisq for each chip are checked to make sure that they are within the cut boundaries (the "DCMS-location", "DCMS-width", and "DCMS-chisq" cuts, respectively.) They are also added to the cumulative histograms.
- *common problems:* By far the most common problem is that some chips, especially the newer SVX4.2B chips, will not clock out data properly when RTPS is turned on and there are not enough channels at pedestal (typically, only a very small number of channels are read out, even if all channels are supposed to be read out). (See also section A.1 in the Appendix.) This will cause readout errors and cause `htwish` to report a "fail" result after testing. In general, it is perfectly safe to ignore this "fail" result, and when running an analysis, `htwish` will know to ignore these errors.

  Note that this means that there will be no data at all for the part of the histogram before RTPS begins working properly, so don't be surprised to see that this data is all 0. (Also, note that since the "low level" is derived from the fit, it might be slightly less than 0. This is just a reflection of the results of the fit, not the actual data.)

5.7 Sparsification and sparsification with neighbors

- *test description:* In these pair of tests, the chips on the hybrid have their read-all flag turned off (unlike most of the other tests, which always read out all channels). In the ordinary sparsification test, the read-neighbors flag is off, and in the sparsification with neighbors test, the read-neighbors flag is on.

Then, a series of different injection masks is used (first, channels 0, 8, 16 ... are injected; then 1, 9, 17 ...; and so forth) so that all channels on the hybrid are injected at some point. With each mask, the chip is read out, and the channels which are read out are checked. If anything unexpected occurs (i.e., `htwish` reads out a channel it didn't expect to read out, or vice versa) then the test fails. If all channels read out as expected, the test passes.

- *histograms:* None.
- *analysis:* The pass/fail result is applied to each channel in the "single-channel sparsification" and "nearest-neighbor sparsification" cuts.
- *common problems:* By far the most common problem is quite simple — the sparsification threshold is improperly set. If it's below pedestal, then all channels will always read out, and if it's above the injected level, then no channels will ever read out. So, check `sparse_hyb.ini` in your appropriate patterns directory, and make sure that "ADC digital threshold" (the sparsification threshold) is set to a value between the pedestal level and the injected level.

  The sparsification test is performed in cell 1, so any known bad capacitors in cell 1 will cause this test to fail. You can mask them out using the features described in section 2.6.

  Occasionally, a hybrid will pass the sparse test but fail sparse with neighbors. This usually happens across chip boundaries, and is indicative of a bad TN/BN bond between the two chips. Fixing this bond will usually get the neighbor test to work properly.

5.8 Risetime

- *test description:* The risetime test is a study of the preamp risetime. How it works is a little complicated. Essentially, the test starts off with a pattern — `hyb_risetime.pat` — which contains the L1A trigger in a fixed bucket and the calinject pulse (that is, the calinject line raised from low to high) a few timesteps before it (a timestep being the unit of time a single character in the pattern represents – this is the length of one clock on the PATT board, or 25 ns for a typical 40 MHz clock). This pulse occurs such a brief time before the L1A that it ends up in the bucket after the bucket the trigger triggers on. So the pattern is run, as usual, and the values in all the channels are read out. Not surprisingly, we get pedestal values, because we're not looking at an injected bucket. However, this will not remain the case. A loop is set up by the test in which, for each successive loop, the calinject line goes high one timestep earlier than it did in the previous loop. Eventually, the inject line goes high in the bucket our fixed trigger looks at. What happens then is that initially the preamp (at least if it's at a slow setting) will not fully register the injected value before its voltage is recorded. As we move further back in the bucket, eventually the preamp does register the full voltage. The results for every channel are recorded, and this test is run for three different preamp speeds (bandwidths), and two different polarities (i.e. both ramp up and ramp down).
- *histograms:*
  - Pedestal: 2D histograms showing the pedestal as a function of number of timesteps the injection has been moved forward. There is one of these for each polarity and bandwidth.

- Pedestal sigma: 2D histograms showing the standard deviation of the pedestal (i.e., the noise) as a function of number of timesteps. Again, there's one of these for each polarity and bandwidth.
- Good pedestal/average pedestal: This entry deserves some explanation. For older SVX4 chips, the results from the tests with polarity 0 (ramp up) and polarity 1 (ramp down) are averaged to produce the data to which a curve is fitted. However, for the newest SVX4.2B chips, the ramp up and ramp down settings no longer produce comparable results, so they cannot be averaged like this. Consequently, the program will only use the polarity 0 (ramp up) results for its curve fitting. There is one of these for each bandwidth setting.
- Good sigma/average sigma: 2D histograms showing the standard deviation for the above histogram. Again, there is one of these for each bandwidth setting.
- Risetime: a 1D histogram showing the risetime as a function of channel. The program obtains this data by fitting a curve to the data (the good pedestal or average pedestal) and measuring the difference between the 10% and 90% levels.
- Fit chisq: a 1D histogram indicating the goodness of the risetime fit.

- *analysis:* The risetime for each bandwidth, as well as the chisq for each fit, is compared to the appropriate values in the "T_r bw" and "T_r bw chisq" channel-level cuts. They are also added to to the cumulative statistics.

- *common problems:* The fitter is a frequent source of problems when analyzing the risetime data. Especially for the newest SVX4.2B chips designs, the risetime for bandwidth 1 is frequently so fast that the fitter can not sensibly fit a curve to the data. Furthermore, since the test can only change the pattern in increments of 1 timestep, the resolution of the test is essentially limited to 1 timestep. Consequently, the risetime histogram for bandwidth 1 will often look odd, with some channels recording 1 timestep and some channels at values less than 1 timestep. But since values below 1 timestep are not really meaningful given the resolution of the tests, they can be safely ignored.

  Sometimes the fitter will fail entirely for no apparent reason; usually merely re-running the analysis will fix this problem.

  Another problem (which is hard to spot; its most conspicuous feature is a warning that a channel failed a sanity check) occurs when the chip is pushed into saturation and a channel drops out of the data. (Since the risetime pattern uses an extra-long bucket, this problem is more likely to occur in the risetime tests.) The simplest way to deal with this is to lower the ramp slope in `rampslope.ini` so that the chip is not operating in saturation. See section A.1 in the Appendix for further details.

5.9 Multihit

- *test description:* The multihit test is designed to ensure that all the chips on the hybrid are capable of handling four closely spaced L1A triggers, and thereby storing four events in their pipeline. Its testing method is quite simple; this L1A-heavy pattern is run several times, and each time the standard data integrity check is performed. This is the only testing method used, but it is sufficient to determine whether or not we're capable of reading out four stored events.

- *histograms:* None.
- *analysis:* None; all that the analysis does is make sure that no readback errors occurred.

5.10 Linearity

- *test description:* The linearity test allows the user to study the response of a channel to different injected charges. It's a bit like the gain test, but rather than measuring the channel's ADC value at two different injection values, it does so over a range of values. These values (low voltage, high voltage, step value, number of steps) are defined at the top of the file `test_linearity.c` and can be easily modified.
- *histograms:*
  - ADC average: a 2D histogram showing the average ADC counts for each channel as a function of the applied voltage.
  - ADC noise: a 2D histogram showing the noise (the RMS variation in the ADC counts) for each channel as a function of the applied voltage.
  - Gain slope/Gain offset: The program fits a line to the "ADC average" data for each channel. These two 1D histograms show the slope and offset (y-intercept) for the fitted line.
  - Gain chisquare: a 1D histogram showing the goodness of the line fit for each channel.
  - ADC distribution: a 1D histogram showing the distribution of the raw ADC counts read back. (Note that this is different than examining a distribution of the "ADC average" histogram, since that will just be a distribution of the average, while this is a distribution of all counts.)
  - Gain raw data: a 1D histogram showing the ADC counts versus voltage for a specific channel.
  - Gain diffs: a 1D histogram showing the difference between the raw data and the fitted line (that is, the residuals) for a specific channel.
- *analysis:* Not used. The linearity test is designed for the histograms it produces for visual inspection, and is not used in the analysis.

5.11 Capnoise

- *test description:* The capnoise tests the effect of varying capacitance on noise. It is designed to be used on the standard SVX chip test board, where different channels are loaded with different capacitor values. If you do not have a setup with varying capacitances, then this test will not be very useful.

  The test also converts the noise results to ENC (equivalent noise charge), as measured in electrons. In order to do this, it has to obtain the ADC counts-to-electrons conversion factor, so for each channel it will also step over a set of input voltage values and measure the resulting ADC counts, then use that data to obtain a conversion factor. (Since this conversion factor might be different for differently-loaded channels, this part of the test is performed once for each channel tested.)

The different capacitances also cause different risetimes, so the capnoise test has the feature of varying the bandwidth settings, as well as the preamp current, for each channel tested to create roughly constant risetimes, so that the noise levels will be comparable.

In order to tweak the settings used in the capnoise test, you'll have to edit `capnoise.h`. The key feature is `channel_array[]`, which defines the list of channels used in the capnoise test. Each element should contain, in order, the channel number, its capacitance (in pF), the bandwidth setting to use for that channel, and the preamp current setting to use for that channel.

There is also an experimental feature to test differential noise (dnoise), which relies on using pairs of channels with the same capacitance and measuring the differential noise between them. It is turned off by default (because we never had sufficient pairs of comparable channels for it to be useful), but to use it, go to the top of `capnoise.h` and uncomment the `#defines` of `TEST_DNOISE` and `SHOW_DNOISE`. Then edit the variable `dnoise_array[]`. Each element should contain the number of the first channel in the pair, the number of the second channel, and the common capacitance (in pF). Note that db files created with `TEST_DNOISE` off are fully compatible if the program is recompiled with `SHOW_DNOISE` on, and vice-versa.

- *histograms:* In addition to the histograms produced, the capnoise test will display some information to the console when it is loaded. First, it will display the calculated ENC conversion factor for each channel used in the test. Then, it will take the ENC-versus-capacitance plot and attempt to fit a straight line to the results, using sets of three points at a time, and will display the results to the console. The histograms are:
  - ADC noise vs. capacitance: a 1D histogram showing the measured noise, in ADC counts, versus the channel capacitance.
  - ENC vs. capacitance: a 1D histograms showing the measured noise, in ENC (equivalent noise charge, in electrons), versus the channel capacitance.
  - ADC/ENC DNoise vs. capacitance (if dnoise is enabled): same as above, but these 1D histograms show the measured dnoise (differential noise) in ADC counts and ENC.
  - ADC vs. voltage: 1D histograms showing the raw results from the voltage scan used to obtain the ENC conversion factor. There is one of these for each channel.
  - Equivalent electrons vs. voltage: 1D histograms containing the same data as above, but multiplied by the ENC conversion factor to show the equivalent number of electrons.
  - ADC noise vs. voltage: 1D histograms showing the noise during the voltage scan, one for each channel. These are mostly provided to investigate if changing the voltage has any effect on the noise.
  - ENC vs. voltage: 1D histograms showing the same data as above, but multiplied by the ENC conversion factor to show the equivalent charge in electrons.
- *analysis:* Not used. The capnoise test is only used to produce histograms, and is not used in the analysis results.

5.12 Currents

- *test description:* This is a very basic test, which simply measures the DUT's AVDD and DVDD currents. A GPIB software/hardware interface is needed; you can learn more about this in section 2.
- *histograms:*
  - AVDD/DVDD currents: 1D histograms showing the overall results of the current measurements.
  - AVDD/DVDD current measurements: 1D histograms showing each individual current measurement.
- *analysis:* The currents are checked to make sure they are within acceptable bounds in the "AVDD current" and "DVDD current" hybrid-level cuts. They are also added to the aggregate histograms.

5.13 Getmap (no charge, with charge, and dynamic)

- *test description:* The purpose of the getmap tests is to test the hybrid's "deadtimeless" performance. That is, they test to see if the performance of the front end is affected by what the back end is doing (whether the back end is idle, digitizing, or reading out).

  The getmap test works by using a pattern with two L1A triggers. The first is a "dummy" trigger, whose only purpose is to start the back end digitizing or reading out; the second L1A is the trigger that we're interested in the data from. The second trigger is stepped, over the course of the test, through the different parts of the pattern: first with the back end idle, then with the back end digitizing, and then through the readout of the different chips. Each time, the pattern is run 46 times (one for each cell), to make sure that all cells are tested.

  The test also changes the number of front-end clocks in the pattern at various times, so that the first pipeline cell tested is not always the same.

  This test comes in three different varieties: the "getmap no charge" test features no charge injection, the "getmap with charge" test features charge injection in a selected subset of channels, and the "getmap dynamic" turns on sparsification (with read neighbors on) and injects into a selected subset of channels.

  These tests actually comprise the majority of the testing time in `htwish`, since it's run three times, and each iteration steps over a very large number of buckets.
- *histograms:* All three variants have the same set of histograms (although the histograms dividing the channels with/without charge are pretty useless for the "getmap no charge" test, since no channels are injected in that case).
  - Number of acquisitions: a 2D histogram showing the number of acquisitions as a function of channel and bucket. Should be 46 for all channels in "getmap no charge" and "getmap with charge", and 46 for injected channels and their neighbors in "getmap dynamic".
  - Bad number of acquisitions: a 2D histogram showing the number of acquisitions in which a channel read out that wasn't expected to, or vice-versa. Should be 0. *Note:*

Bad capacitors will show up in this plot (even if masked out in testing), and this histogram is used to automatically mask them out so as not to affect the "P_getmap ped residuals" and the "D_getmap N_acq" cuts.

  – Min/average/max observed pedestal: 2D histograms showing the minimum, maximum, and average pedestal as a function of channel and bucket.
  – Pedestals with/without charge: 2D histograms showing the pedestals for channels with charge injection and those for channels without charge injection. (Note that the channels are renumbered in these histograms, so that, for example, channel 60 in the "pedestals with charge" histogram is actually the 60th injected channel, not the 60th channel overall.) These histograms are also displayed for the "getmap no charge" test, but they don't actually mean anything in this case.
  – Noise: a 2D histogram showing the noise as a function of channel and bucket.
  – Noise with/without charge: 2D histograms showing the noises for channels with and without charge injection. (Like the pedestals, the channels are also renumbered here.)

- *analysis:* Note that most of the cuts come in "P", "C", and "D" varieties, which refer to the "getmap no charge" (i.e., pedestal), "getmap with charge", and "getmap dynamic" tests, respectively. The C and D cuts also come in two different varities: "noc", which refers to channels with no charge injection, and "c", which refers to channels with charge injection.

  – N_acq: a channel-level cut to make sure the overall number of acquisitions is correct. (Bad capacitors are automatically masked out for the D_getmap N_acq cut.)
  – pedestals: a chip-level cut checking to make sure that the overall pedestal is within acceptable levels.
  – ped swings: a channel-level (for P) or chip-level (for C or D) cut to make sure that the difference in pedestal during different buckets is not too large.
  – ped residuals: a channel-level (for P) or chip-level (for C or D) cut checking that the pedestal residuals of individual channels are not too large.
  – noise: a chip-level cut to make sure the noise is within acceptable levels.
  – ped diff (C and D only): a chip-level cut to check that the difference between injected and non-injected channels is within acceptable bounds.

  All of the above (pedestals, noise, minimum pedestal residuals, maximum pedestal residuals, pedestal swings, peak noise, and pedestal differences for "getmap with charge" and "getmap dynamic") are added to cumulative histograms. For "getmap with charge" and "getmap dynamic", these are accumulated for the uninjected and the injected channels separately. Note that for the residuals and swings for "getmap no charge" are on a per-channel basis, while everything else is on a per-chip basis.

- *common problems:* There are two common problems that we've seen in getmap. The most common is when only getmap dynamic fails. This is almost always because of a bad capacitor, since getmap cycles over all cells, so any bad capacitor (in an injected channel) is going to cause problems. You can mask them out using the features described in section 2.6.

  If all three getmaps are having problems, this is probably an indication that the cell ID isn't what was expected. The cycling pattern used by getmap is very complicated, and so the computations used by getmap to determine the expected cell can sometimes fail. Try to determine the bucket at which it begins having problems, and then edit `test_hyb_getmap.c`, find the area where `cycle_offset` is determined, and fix it appropriately.

If your analysis is failing for "getmap with charge" or "getmap dynamic", it might be because the channel mask in `getmap.ini` or `getmap_dynamic.ini` is not as expected. Check the top of `fetch_hyb_getmap.c` and make sure that the channel mask defined there matches what's in your appropriate .ini file. Also, if you've changed the total number of injected channels (27 in the old patterns, and 32 in the new patterns), you'll also need to change the value of `nchan_mask_c` in `analyze.tcl`.

6 Working with `htwish` Code

`htwish` is not a piece of commercial software, and occasionally a user of `htwish` will want to alter the program's source code. Fortunately, the only changes that will ever need to be made are utterly trivial. The first part of this section will describe alterations you'll probably want to make to `htwish` at some point, and how to make them. The second part is written for those who are simply curious about some of the more important and/or problematic sections of `htwish` code.

6.1 Changing the code

Changing the set of tests `htwish` runs:

One of the most common actions you'll want to take when getting `htwish` to work with your test setup is to change the list of tests `htwish` performs on a hybrid. This is especially true when you're trying to figure out why a hybrid is behaving as it is under a given test, in which case you'll want to run only that test, and not all the others! The solution here is very simple: go to the file `run_tests.c`; near the top of the file is a list of `#define`d tests, which account for all the tests `htwish` is capable of running on a hybrid. Simply comment out the tests you don't wish to include in the test suite, re-compile the program, and you're done.

Changing the set of tests used in `htwish`'s pass/fail analysis:

Also important is the ability to alter the tests `htwish` requires to run a pass/fail analysis on a hybrid. This is especially useful if, for example, you want to gather aggregate statistics on pedestal values for a large number of chips, which requires analyzing them, but you don't want to have to do a full analysis. In order to do this, you'll need to open the file `testdisp_globals.tcl`. At the top of this file is the full list of tests `htwish` uses for its analysis; simply make sure that the tests you don't want `htwish` to use are commented out. Note that while C/C++ employs `//` to comment out a line, Tcl uses the `#` character.

Removing a cut from the analysis:

If you wish to keep a test, but are having problems with a specific cut that that test uses, you can edit `combine_anal_results.tcl` to remove that cut from the final analysis. First, find `badcap_tests`, `badchan_tests`, `badchipq_tests`, or `badhyb_tests` (depending on whether this is a capacitor-level, channel-level, chip-level, or hybrid-level cut), then remove the test name from the list after it.

Altering the cut values:

Finally, if you want to keep a test but change the range of cuts it uses to evaluate a chip or hybrid, go to the file "analyze.tcl". In this file, every test's criteria are found in a function with a name of the form "anal_proc_<testname>". If it's a test which employs cuts, the cut values will be listed near the top of the function's code. Simply change their numeric values, and you've changed the analysis criteria. (Read the test description in section 5 for details of what the cuts do.) The cut values currently used are derived from our preproduction batch of hybrids, but it's entirely possible that a hybrid you consider "good" might not meet those standards. In that case, feel free to change the cuts and let us know.

Calibrating `htwish`'s voltage output:

A few tests, as described earlier, operate by varying the voltage sent to a chip or hybrid. An unfortunate fact of life is that the voltage `htwish` is told to send out to the DUT is **not** necessarily the voltage seen by the DUT; issues such as the impedance on the hybrid board and the configuration of the specific DAC interfere with this. To remedy this problem, `htwish` has a function called `voltage_to_vcal` which is designed to take the value of the physical voltage the user wants the chip/hybrid to see and convert it to the voltage the computer will need to send out to accomplish this. To be absolutely sure that your DUT actually experiences the voltage it's supposed to, I suggest you measure the linear relationship between the computer voltage "VCAL" and DUT voltage. Then enter this relationship into the `voltage_to_vcal` function, located in the file `voltage.c`.

Note that `#defines` are used to differentiate between SVX4 and SVX4.2 chips, as well as between single-chip boards, L0 hybrids, and standard hybrids. Make sure you change the right ones!

Suppressing the display of the quality tree:

On line 215 of `anal_utils.tcl` is the command `display_cuttree`. Simply comment this out if you don't want `htwish` to display the quality tree after it runs its analysis.

6.2 Important code

In addition to these specific changes you can make to the code, there are a few other functions you should be aware of in order to understand how `htwish` is written:

SVX Test Functions:

Many functions used by `htwish` come from a set of libraries written a few years ago specifically for the purpose of working with the SVX chip test setup. As they have been documented before, I will only briefly touch on them. For more info, look at Maurice Garcia-Sciveres' and Dipanjan Ray's 1998 paper "The PATT Chip Test Setup", available at
`www-cdf.lbl.gov/svxii/Hardware/patt02.ps`.

`appini_change`: This is normally called by `load_inifile` in `htwish`. It takes the .ini file whose name without the extension or the (default) directory is passed to the function, as well as an array of initialization keyphrases and values which it uses to change the initialization pattern given in the file.

`addpat("patfile")`: This will set up the pattern found in the file `patfile.pat`. If there's already a pattern in memory, this pattern will be added to the current pattern.

`bs_pad_level("")`: This insures that all lines controlled by the pattern currently in memory are controlled for the same length of time (or number of characters).

`set_pat_bit / reset_pat_bit`: These functions will set the control character in position N in the current pattern high and low, respectively. The most obvious use is found in the risetime test, when the cal_inject line is set high progressively earlier in that test's pattern file.

`out_go / out_run`: These are the functions used to run the pattern from the computer's memory on the chip hybrid using the PATT02 / PATT03 hardware setup. However, you'll only see `out_go` in the code; the program is made to work with the PATT02 board by accessing the `hybtest.h` header file and `#defining` `out_go()` as a macro which calls `out_run()`. This can be a source of confusion if you're not keeping an eye on it.

Additionally, there are other functions written specifically for `htwish` which complement these standard test functions. Here's a quick overview of them.

`load_inifile.c`: `load_inifile`

```
int load_inifile(char *inifile, Hybrid_type htype, int nchips,
                 int lastchip_id, char **parnames,
                 unsigned int *parvals, int npars)
```

`load_inifile` is used to initialize the chips on the hybrid. It will open up the .ini file specified by the `inifilename` variable, taking its directory to be the one specified by a call to `set_path`. It is not, however, strictly limited to initializing the chip using the values in the .ini file. Its last three parameters are important to notice; by passing `load_inifile` an array of initialization parameter names, an array of initialization values, and the number of values in the array, you can allow it to modify the initialization pattern it sends to the hybrid chips. This is extremely useful, since in many of the tests, loops are set up in which the hybrid's chips are re-initialized with different values for one or more parameters every iteration. Please note a potential pitfall: the initialization parameter name has to be a character-by-character match to the name in the .ini files, otherwise unpredictable results will occur!

`hyb_parse_data.c`: `hyb_parse_data`

```
Hybrid_errstat hyb_parse_data(int* word, int* bit9, int nchips,
               int event_start_point, int expected_chip_number,
               int expected_cell_id, int* channels_to_read)
```

This is the function which performs the data integrity check. After an event is read back from the hybrid and stored in a buffer, this function will take the buffer (`word`), as well as a number of `htwish`'s expectations, and see if the readback data live up to those expectations. Specifically, the program will provide `hyb_parse_data` with the byte number in the buffer in which the event begins, as well as an array (`channels_to_read`) which lists all the channels on the chip which it expects to have been read out. It will also provide it with the `bit9` buffer, which contains the value of line 9 on the standard LBL hardware test setup (if you don't have this hardware setup, refer to section 2 to find out what to do). The value of line 9 is either priority-out (in the serial line test) or odd-byte data valid (in all other tests). Some of these parameters assume defaults; `event_start_point` set to -1 will assume that the event of interest was the first one stored in the buffer (not the case in the getmap tests), and `channels_to_read` set to NULL will assume that all were supposed to be read out. The value returned by `hyb_parse_data` is an error structure, `Hybrid_errstat. If hyb_parse_data` finds a disparity between the readback event and the program's expectations, then it will supply the error structure with a nonzero value, otherwise 0. Essentially, `Hybrid_errstat` contains the data needed to fill the error table. See section A.4 of the Appendix for the full error table.

6.3 Code structure

This is a very brief overview of the most important components of the `htwish` source code:

test_*.c: These files implement the method by which `htwish` runs a test; please note that `pipeline_ped_map.c` is an exception in that it does the same thing as the test_*.c files, but lacks the name format!

fetch_*.c: These files implement the method by which `htwish` loads in data from a .db file during the analysis GUI phase of the program.

run_tests.c: The main structure of the C-written portion of `htwish`; this file defines which tests are run by the program when in test mode.

analyze.tcl: For those tests which are used to evaluate a hybrid, each test's analysis criteria (cuts, e.g.) are found in functions defined in this source file.

combine_anal_results.tcl: This is the file which defines the hierarchical structure of cuts and pass/fails employed by `htwish` in its evaluation of a hybrid.

A Appendix

A.1 Troubleshooting / comments

There are a number of things that can go wrong during the running of `htwish` (as I've learned from experience). Here, I attempt to draw on my own experience and the experience of others to point out areas where it's easy to go wrong.

Occasional segmentation fault:

I've noticed that occasionally, when running `htwish` in test mode, that a segmentation fault will occur in the opening seconds of testing. Simply re-running the test will generally solve this problem.

DCMS failure:

During the testing phase of `htwish`, we've noticed that often DCMS will fail whereas all the other tests will be passed with the hybrid. Apparently, if many channels are injected, so that the chip is confused into thinking that the injected channels are actually the pedestal, some chips will clock out junk. This problem seems to be more common in the SVX4.2 design.

In general, we've decided that this is not a real problem, since this is a condition not likely to be seen in normal operation. Consequently, if testing reports a DCMS failure due to this, you can safely ignore it (and when you run an analysis, `htwish` will ignore these errors).

Chip saturation:

The SVX4 chip, and the SVX4.2 design in particular, has a known issue that if you operate the chip with a large number of channels in saturation, one of these channels (typically channel 118, occasionally along with a neighboring channel) will drop out and read 0 instead of 255. Needless to say, this behavior will cause a large number of errors. (Since the chip is not normally expected to operate in these conditions, it's something we've decided not to worry about.) To avoid this, adjust your parameters in the .ini files to make sure that the chip is not running in saturation. Be especially careful with the risetime test; since the risetime test uses an extra-large bucket, the charge is integrated over a longer amount of time, so that saturation can occur in the risetime test even if all of the other tests are OK. To avoid this, the simplest way is to lower the ramp slope value in `risetime.ini`.

BerkeleyDB errors:

Sometimes, things will go wrong with the database files, and you will experience problems in loading them. For example, if a memory fault occurs while `htwish` is accessing a .db file in its analysis GUI mode, then all subsequent attempts to create .db files will result in an error message referring to a "pinned block" I/O problem. The remedy is simple, but not obvious: go to your db support directory (the one with the `DB_CONFIG` file in it). Delete all the files in it that begin with a double underscore. Now you should be back in business!

Simultaneous access of database files:

The old version of BerkeleyDB used in `htwish` is not always the most stable. One problem with it is that if you're running tests on a hybrid on one computer, and you open `htwish` in analysis mode on another computer, the attempt to access the currently-open database file can result in a number of unpleasant results, up to and including crashing `htwish` on **both** computers.

The new automatic `run` script addresses many of these issues, by creating a simple lock file so that `htwish` will not attempt to look at an open database file, so you should try to use it whenever possible. If this is not possible, then do be careful about accidentally accessing an open database.

Incorrect number of "junk" bytes:

In setups like our PATT setup here at LBL, there are a number of junk bytes at the beginning of the data. With the `LBL_DAQ` option set, `htwish` will automatically eliminate these junk bytes. However, the number of junk bytes is sensitive to timing issues (cable length, etc.), so if this number changes, it can create a large number of problems.

To change the number of junk bytes at the beginning of the readback data, edit `hyb_parse_data.h` and modify `JUNK_BYTES_START` near the top of the file. If you are using flat ASCII output, then you'll also need to modify `lbl_daq_strip.h` similarly.

You may also have problems with getmap in this case, since the second event may start at a different point, so you'll have to modify `test_hyb_getmap.c` and change `EVENT_START_POINT`.

Entry scripts:

As mentioned elsewhere in this document, there are a number of scripts in the `htwish` code directory (most notably `htest.tcl`) which have a .tcl extension and contain Tcl code, yet appear to be standard Unix scripts based on their first line. Their first two lines always follow the same format:

```
#!/bin/sh \
exec <some tcl-based program>
.
.
.
<Tcl code in the rest of the file>
```

So which is this — a .tcl file or a Unix script? Actually, it's both, and the reason is because whereas the backslash is a line-continuation character in Tcl, it means nothing where it's placed in a Unix shell script. So what happens is that the first command this Unix script executes is whatever Tcl-based program is "exec"-ed on the second line of the script. But the first two lines appear to be a single commented-out line to the Tcl interpreter, so when our Tcl-based program runs it can use this script file as a Tcl code source. Incidentally, try not to attempt anything other than the orthodox (Unix script) method of running `htwish` when it's in test mode. There's a lot of sleight-of-hand that goes on in these scripts, and some scripts even have the same name as Tcl functions, so it's easy to make mistakes when trying to run the program from the command line.

The load_inifile function:

The `load_inifile` function, as mentioned in section 6 of this document, can easily cause unpredictable results if its parameters are not passed correctly. Specifically, when trying to change the parameters to the initialization file you refer this function to, make sure that in your parameter names array you have the names correctly spelled out — to the character, and case-sensitive. Otherwise you'll confuse the function, and botch the initialization of your hybrid.

The obtain_inifile_mask function:

This function, implemented in `obtain_inifile_mask.c`, will take a standard .ini file, reads in the mask it sees at the top of the file, and fill in an array to reflect that mask. If the .ini file is formatted incorrectly, then the function will fail. This shouldn't be a problem as long as you edit existing .ini files carefully.

out_run / out_go:

These two functions, implemented as part of the standard SVX test library, essentially do the same thing, except that the first runs a pattern with a PATT02 board and the second runs a pattern with a PATT03 board. If you mix them up, you won't be able to run patterns on your hybrid. This problem becomes especially tricky since in the PATT02 version of the code, it appears to be `out_go` that's called all the time, until you realize that there's a macro which redefines `out_go` to be `out_run` in `hybtest.h`!

`htwish` **ported to a system with a different clock frequency:**

The numerical ADC value read out from a hybrid's pipeline capacitor is directly proportional to the PATT board's clock frequency. If `htwish` is ported to a system with a different clock frequency, then large-scale changes will need to made: to the analysis cuts, to the sparsification thresholds, etc.

A.2 Switching between SVX4 and SVX4.2 chips

`htwish` supports testing and analysis with both SVX4 and SVX4.2 chips. Note that "SVX4" refers to the prototype SVX4 and "SVX4 v2" (not to be confused with SVX4.2!) chips produced before May 2003. SVX4.2 refers to the production chips produced after May 2003, which come in two types – SVX4.2A and SVX4.2B. There are no major differences in usage between the SVX4.2A and B designs.

As noted in section 2, in order to compile `htwish` properly, you must give it either the `SVX41` or the `SVX42` flag. If you wish to switch to testing the other chip type, you must recompile `htwish` with the new flag.

SVX41 means the following: 190-bit initialization bitstream, both polarities tested in the risetime test (see section 5.8 for more details), chip ID offset by 0, cell ID offset by 192. SVX42 means the following: 192-bit initialization bitstream, only ramp up tested in the risetime test (see 5.8), chip ID offset by 128, cell ID offset by 0.

Since the performance of the SVX4.2 chip is different in many aspects from the SVX4 chip, essentially all of the major cuts have been changed for the SVX4.2. If you wish to switch between analyzing different types of chips, you'll need to change the cuts appropriately. This requires editing `analyze.tcl`. Most of the old SVX4 cuts are still present (but commented out) in the appropriate sections. You'll also need to change the number of injected channels which is stored at the top of `anal_getmap_charge` in `analyze.tcl`; it's 27 for the old patterns and 32 for the new patterns.

A.3 A note on notation

The chip itself numbers the cells from 1 to 46 — that is, if you look at the raw data coming from a chip or hybrid, the cell IDs returned will be in the range 1 to 46 (plus the cell ID offset, if any). However, for historical reasons, `htwish` numbers the cells starting with 0 instead, that is, from 0 to 45. This can result in some confusion. Please remember that `htwish`'s output will have the cells numbered from 0 to 45, and it expects the wafer probing data (if any) to have the cells similarly numbered, but if you give it bad capacitors on the command line (as opposed to through `run`, which will automatically perform the conversion), they should have the cells numbered from 1 to 46.

`htwish` numbers the chips in a hybrid as you would expect — chip 1 is the first chip in the readout, and chip N (where N is the total number of chips in the hybrid) is the last chip in the readout. However, be aware that in a normal hybrid, chip 1 (the first chip in the readout) is actually the rightmost chip on the hybrid, and the last chip in the readout is the leftmost chip on the hybrid. (You can look at the priority-in and priority-out lines to confirm this.)

The channels are always the same — both the chip and `htwish` will number the channels from 0 to 127. However, for a multi-chip hybrid, `htwish` will number all of the channels in sequence, so channel 128 is actually channel 0 of the second chip, and so forth.

A.4 Error table

When `htwish` parses the data read back from the hybrid during testing, it assigns a number from 0 to 11 to the data. Any value other than 0 is an error, and means that the test will be displayed as “fail” at the end of testing and that an error table will be available if those test results are viewed in the GUI.

The meaning of the error codes is as follows:

0 Data was OK.
1 Unexpected number of bytes read back from hybrid.
2 Possible priority passing error: With the `LBL_DAQ` option set, `htwish` expects there to be two junk bytes (with OBDV not valid, of course) between each pair of chips on a hybrid. If these are not present, it will generate this error. (With `LBL_DAQ` not in effect, this error should not appear.)
3 Unexpected chip ID byte: The chip ID received was not the expected chip ID.
4 Invalid chip ID byte: The chip ID received was an invalid chip ID (i.e. the value was less than the chip ID offset).
5 Missing chip ID byte: No chip ID byte was found in the data.
6 Unexpected cell ID byte: The cell ID received was not the expected cell ID.
7 Invalid cell ID byte: The cell ID received was invalid (i.e. the value was less than the cell ID offset, or the cell ID was greater than 46).
8 Unexpected channel ID byte: A channel ID was received that was not the expected channel ID.
9 Invalid channel ID byte: A channel ID was received that was invalid (i.e. greater than 127).
10 Last channel didn't come out.
11 Bad odd byte data valid: OBDV should be 1 for chip ID and channel ID, and 0 for cell ID and channel data.

A.5 How the .pat / .ini files work

This is described in section 6.3 of Maurice Garcia-Sciveres' and Dipanjan Ray's 1998 paper "The PATT Chip Test Setup", available at
`www-cdf.lbl.gov/svxii/Hardware/patt02.ps`.

A.6 Using `htwish` with `htmon`

`htwish` has features which allow it to be used in conjunction with `htmon`, a program written by Igor Volobouev for monitoring hybrids during burn-in. When used with `htmon`, `htwish` uses a special test type, "periodic", which support tests run periodically during the burn-in, as well as two additional tests – "bias currents", used to measure the bias currents during burnin, and "readout check", used to periodically check that the hybrid is reading out properly.

Most of the features used for interoperation with `htmon` are specifically hard-coded to work with `cdfsg1`, the machine used as the burn-in stand for the SVX3 testing. If you are interested in adapting these features for your own use, please contact Igor for more information.

A.7 Igor Volobouev's documentation (summary)

There are a number of .txt files in the code directory which Igor had written when he wrote `htwish` for the SVX3 test setup a couple of years ago. Some that you may want to take a look at include:

changing_db_dir.txt: basically describes how you can change the directory in which you store your .db files; also complements the information in section 2 on how to set up your .db directories.

adding_new_test.txt: the filename pretty much describes what it is. Not that you would want to add a test yourself, but this document can give you a better grasp of how `htwish` code is structured.

histo_tcl_api.txt: describes the interface between `htwish`'s Tcl front end and the HistoScope functions. Mainly for the curious.

new_histo_commands.txt: this describes additional HistoScope functions added to the standard library. Also mainly for the curious.

hs_calc.txt: this describes the `hs_calc` function, a function added to perform various calculations on HistoScope histograms. Also mainly for the curious.

analysis_cuts.txt: provides a nice explanation of the hierarchical method of hybrid evaluation `htwish` uses.

A.8 Contact info

If you have any questions about any of this, get in touch with Paul Lujan at LBNL. He can be reached at `pjlujan@lbl.gov`, or at the CDF lab, 510-486-7305. You can also try to find John Freeman at `jcfreeman2@lbl.gov`.