L1 DCT Upgrade Production Teststand Requirements

Version 0.0

1 Introduction

The production teststand at SLAC for the L1 DCT upgrade testing before the final system integration tests need to cover four specific areas:

- ZPD board tests.
- TSF board tests.
- GLT board tests.
- Interface board tests.
- DCT/GLT component connection interface tests.

The basic goal of the teststand is to fully qualify individual boards and their interface connections. This requirement document is intended to only address these production board tests, while the TSF and ZPD will also have their initial test systems at Manchester and Harvard respectively which need to be specified separately. Another area with considerable uncertainty is the initial interface board testing when TSF/ZPD are not yet available.

Much of what we could write down now are based on our experience of previous teststands. The teststands typically contain a server running on the ROM to actually communicates with the trigger boards, while a GUI running on the UNIX side transmits user commands from the GUI to the ROM server and receives the return data from ROM server to display in the GUI. Another scheme of using command language diagnostics to directly interact with the ROM is probably a simple route for a small sets of matured system checkouts, while the board debugging teststands have a wider range of (and forever changing) tests which we cannot expect all users be familiar with the command language. The original approach with the GUI is probably still the most effective for an average board debugging user.

2 Hardware Requirements

The concern here is that the bulk of testing activities are expected to peak at the about the same time during April-August/02. In order to not slow down the activities in each area, the ideal arrangement is to have 3 separate teststands. With just two crates, it would be in fact quite crowded as at the height of the testing, there are 30 TSFs to go through which can easily lock up one system solidly for a long time. The hardware availability status are as follows:
<table>
<thead>
<tr>
<th>Name</th>
<th>Frontend crate</th>
<th>DAQ crate</th>
<th>ROM</th>
<th>FCDM/FCPM</th>
<th>DCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSF</td>
<td>(from IR2 or procure)</td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>??</td>
</tr>
<tr>
<td>ZPD</td>
<td>from LBL</td>
<td>from LBL</td>
<td>from LBL</td>
<td>from LBL</td>
<td>from LBL</td>
</tr>
<tr>
<td>GLT+Interface</td>
<td>In Dataflow Lab.</td>
<td>yes</td>
<td>ROC13</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

The GLT frontend crate is in fact a “hacked-together” crate with borrowed PSU. The eventual commissioning with the 3 crates of new DCT system needs more front-end crates any way. We only have two full power crates (IR2 spare and LBL teststand) so that at least one more full power crate to procure. in any case we need to build more frontend J1/J2 backplanes and obtain more DIRC J3s to assemble additional frontend crates.

3 The Basic Software Utilities

Fast Control Command Interface: This is the ROM level code which interprets each of the fast control commands and executes using basic dataflow utilities. The commands can carry different numbers of arguments, while some of the binary commands such as Run/NonRun mode or Continuous/SingleShot play are better to carry the state names in the accessor function names instead of having 0/1 as arguments. The interface should wrap all dataflow internal protocols such that externally exposed arguments are simple integers. This should be a basic utility extending beyond the teststand use, e.g. eventually the formal calibration processes should also be based on this utility.

The UNIX/ROM Communication Utility: This provides a general communication protocol to transmit commands from UNIX GUI to the ROM server. It also provides a return path for data from the server back to the GUI for display. This is the unfortunate extra layer needed compared to the command language diagnostics directly talking to the ROM. Two tricky issues typically emerge in this part of the code:

- Many operations need a handshake between the UNIX side and the ROM side so that the UNIX side can wait for the ROM execution to complete before carrying on to the next task. The mechanism to catch the return data from the ROM is a subset of such handshake.
- The UNIX side can issue a ‘compound’ command such as ‘Memory test’ which in fact consists of many lower level commands of write/read/check. There is always a dilemma as to whether interpret the compound commands into many more basic commands then send the ROM one by one, or just let the ROM to interpret everything. The latter is clearly more efficient in execution given the difficulty in UNIX/ROM handshake. One can probably consistently follow the principle of minimal interpreting on the UNIX side.

The UNIX GUI Displays: The GUI displays can be controlled by tcl as before (or other means which can do better). This need to allow each type of board having their
own GUI panels. This is probably sensible for handling control commands and simple item displays while more complex table displays can only be done easily on the ROM window.

The Data I/O Utility: The tests can involve input/output data of various forms (flat files, XTCs, binaries etc.). It will be beneficial to standardize the data I/O for each type of data, and standardize their storage locations. Several TC classes may need to be programs for the various boards.

The Command Macro Utility: One of the most powerful utilities for very quickly establishing a canned test with flexibility was the Macro utility in the GLT teststand. It essentially contains the capability of just interpreting a few basic commands:

- **writeCSR** [CSRnumber] [Value]
- **writeMemSingle** [Block-addr] [secondary-addr] [16/32 bit modes] [Value]
- **writeMemRange** [Block-addr] [start addr] [end addr] [16/32 bit] [blockwrite/1by1] [File]
- **readMemRange** [Block-addr] [start addr] [end addr] [16/32 bit] [blockread/1by1] [File]
- **StartPlayBack**
- **CalibStrobe**
- **Synch**
- **SetPrintOption** [Option Value]

Almost all test setup can be accomplished with a stack of commands in this group listed in a flat file which can be edited very easily. The original implementation in GLT just used the simple number codes for the commands, while the L1 diagnostic code should already have equivalent command text interpreter. Experts should be able to easily add some board specific compound commands to the command interpreter.

4 The Dataflow/Online Code and MC Simulation

We of course need a standard set of online dataflow code for the various new boards. The TSF configuration data format might change, but the TSF TC should be the same as before. The TSF digi access also need to handle 5/6 bit LUT differences. There is a completely new set of ZPD configuration TC/code, event data TC/FEX, Oep/Online Digi code to be written. GLT code has the issue of certain objects are available in the old system but not in the new system (or vise versa), which need some version handling.

The ultimate algorithm tests need full Monte Carlo simulation events to seed the input and check the output against. This will take quite some development time and need to be pursued in parallel. In particular, the data format (a TC ?) for the input/output diagnostic memories need to be designed at a rather early stage.

5 The Basic Common Tests

The test in this section are common tests for all boards which can be done with a uniform panel control taking advantage of the best features from the various teststands and avoid
duplicated efforts. All test panels should have C-link/D-link A/B slot mask selection control to enable users to choose which board to talk to.

**Fast Control Tests:** This should contain all fast control commands with panel control for the additional arguments and output. One should be able to also automatically repeat a command for N times. This is very useful when debugging boards using a scope.

**CSR Read/Write Tests:** The old teststand used to have a standard panel to allow you write specified value to a given CSR. However, the meaning of the bits etc. are typically difficult to remember unless one is using it everyday. It may be beneficial to program the panel with explicit control for each useful bit of each CSR and label by their mode meaning rather than the raw value. However, depending how many memories are on board, this explicit styled control may well have to be under a board specific panel.

**Single Address Memory Read/Write Tests:** This should contain panel control for selecting block-address, secondary address, and download/readback single values for one location in any memory. One should also be able to select 16/32 bit I/O modes and perform read memory and increment address.

**Block Memory Read/Write and Macro Tests:** This should contain panel control for selecting block-address, secondary address range, 16/32 bit modes and download/readback a range of addresses in a memory to/from a flat file. It should also be possible to select block R/W instead of single address R/W. One should also be able to execute a selected Macro file containing a stack of commands for the setup of a canned test.

**Automated Board Memory Tests:** Although all boards have very different types of memories, it should still be possible to share a common memory test utility. There should be various test patterns available, e.g. All zeros, all ones, random numbers, variations of “walking 1” patterns (set bit shifts as address increase/decrease) etc. The walking 1 patterns should have periods seeded off a prime number to avoid repeating patterns for addresses at full multiple of 16 apart. The memory tests can be specified in a memory test option file with one line per block of memory. The specification should include

- Memory type (16/32 bits).
- Memory block address.
- Memory start/end secondary addresses.
- Memory mask period and a list of masks.

The last item needs some explanation. Some memories only implemented a subset of bits in a word to be active bits, and the unused bits may not be writable. One therefore needs to tell the memory test program a mask for the active bits so that
only the readback of the active bits are checked. However, it is quite often that these memories do not have a uniform mask everywhere. e.g. a piece of data memory contains some data from 16 ticks and each tick of data occupies 5 consecutive 16 bits words which have 12, 12, 10, 10, 7 active bits (assuming the lowest n bits), then the memory can be specified as Mask-period=5 followed by FFF FFF 3FF 3FF 7F as the masks. It should also be noted that sometimes the same range of addresses are controlling memories with different physical widths (e.g. the GLT LUT 20 bit words are simultaneously controlling counts for objects of 16, 20, 10 bits widths). When one started to test addresses beyond the physical width, the loaded content would wrap down to the physical region so that test for contents of in the physical range could apparently fail. This requires that the users have to carefully construct the memory test option files to avoid such wrap. This may mean that you cannot test all memories of a board in a single test, but the full memory test for the board need to be done in a few separate tests. A multiboard operation mode should also be considered to allow simultaneous pattern download to several boards, while the readback check probably still need to be done board by board.

6 The Algorithm/DAQ Tests

The algorithm tests are likely to be different from board to board and therefore probably have to be controlled from board specific panels. Although there are clearly many differences in what kind of configurations are needed, and what kind of input, output data need to be dealt with for the algorithm tests, the basic operation steps are probably nonetheless similar, like a calibration process.

1) Configure the system.

2) Start event loop: read in one event from test data file.

3) Set system to non-run mode and download input data to the input end diagnostic memory for this test and set the input/output memories to single-play/record respectively, and system into run mode.

4) Hit start-play followed by an L1Accept at a fixed delay after start-play.

5) Upon Level-1 accept, the DAQ event readout will take place and FEXed TC data can be optionally stored to disk.

6) Set system to non-run mode and read back the relevant output diagnostic memories.

7) Check output diagnostic memory against the test data predicted output. Log errors into a log file.

8) Go to step 2) for another event if the loop has not reached event limit.

9) End of event loop: summarize test results in log file.
The DAQ test in the middle is optional and would be convenient to have the tests being able to check the output for both DAQ buffers as well as the diagnostic memories.

The algorithm tests should be have test samples with different levels of complexity, starting with the simplest patterns (may be in conjunction with simple configurations of e.g. LUT) stepping through different regions of the detector. One simple example may be to play a fixed pattern from the ZPD receiver to the MegaBus and check in the algorithm FPGA input diagnostic memories to see if they received exactly the same pattern from the MegaBus. The complex end of course will be the full BB MC simulation events and results are several stages are checked against MC prediction. We ultimately need to run probably at least a few thousands of events to quality a board. We also need to have test samples with >4 events spaced out in the input diagnostic memory and generate multiple L1Accepts to test the overlapping event and 4-buffer busy effects for the DAQ. The output vs prediction algorithm check code need to have a very well formatted display of board output and prediction side by side, so that one can examine troubled events with single shot detailed displays.

7 The Interface Tests

When we eventually have all the different types of boards in hand, we should be able to perform the full chain interface tests. These tests also need to run through large number of events to test the stability of the transmission as well as. Some of the major tests:

- TSF→TSF neighbor tests. This may be setup as a 3 module system and swapping the center module as the one being tested.
- TSF→TSFi→BLTi→BLT test. Each TSF, TSFi needs to be swapped in/out for the tests.
- TSF→TSFi→ZPDi→ZPD test. Each TSF, TSFi needs to be swapped in/out for the tests. Preferrably to have a 3 TSF/TSFi system to feed 3x3 inputs to fully populate one ZPD/ZPDi and swap in/out each ZPD/ZPDi.
- BLT→BLTi→GLTi→GLT test. This should be a simple test to just verify new GLTi.
- PTD→PTDi→GLTi→GLT test. This should be a simple test to just verify new GLTi.
- ZPD→ZPDi→GLTi→GLT test. This test needs swapping ZDP,ZPDi,GLTi in turn.

This is probably ideally achieved with the teststand which can configure several different types of board in the same crate. The real dataflow platforms may have trouble dealing with e.g. ZPD and GLT in the same crate. The test procedure steps may be inherently similar to the algorithm tests, although it is now dealing at least two different types of boards simultaneously. As can be seen that there are a large combinations of tests for the interface which need to be done so that confining the interface tests to one crate is an important goal for the teststand resource sharing. One could in principle drive two frontend crates together using both A and B links (GLT/EMT tests was done this way), but hopefully that won’t be necessary.
Some special interface tests with the EMT to GLT path is also needed. The TSFi GLINK receiving of DCH data is potentially rather difficult to test in the Lab. It may need some special external test hardware to accomplish. For the commissioning system, the best test is of course the new TSF vs old TSF data direct comparison if they can be taken with split fibers simultaneously. However, we cannot leave leave some trivial problem only to be found then with 30 TSFi’s built.