ALEPH 90-9
TPCGEN 90-1
25.1.1990
J. Fagestrom

TPD36 - A TEST PROGRAM FOR THE TPD MODULES

Jan Fagerstrom

CERN, December 1989

Abstract

This document describes the TPD test program TPD36. It gives an introduction and includes a user guide as well as a software description.

CONTENTS

1. INTRODUCTION

1.1 General information

TPD36 is an interactive test program for the TPD modules in the TPC. It's purpose is quick and effective detection of errors in the hardware (i.e. in the TPD FB modules) and to display the detected errors in a simple way. The program tests the different registers and memories in the TPDs and also the DOLIST- and Read Valid Data operations.

1.2 Program structure

The program consists of two major parts: the user interface, error decoding and display part running on the VAX and the test routine part which runs on the TPPs in the different sectors in the TPC. (This means that the program can not be used in parallel with other activities on each sector since it occupies the TPP!).

The part on the VAX loads the tests and some test parameters (which tests should be executed etc) down to the TPPs. When the tests are done it reads the results back from the TPPs, interprets the errors and displays the outcome on the screen. The VAX part uses UPI to build up the user interface.

The TPP part (i.e. the tests) receives the test parameters which have been set on the VAX and executes the tests according to them. Detected errors are put in the result variables which can be uploaded to the VAX and decoded.

2. USER GUIDE

2.1 Start

TPD36 runs under the switcher in any partition. The program can be found if selecting "Tests>" on the partition controller menu and is called "TPD test 36". See figure 1!

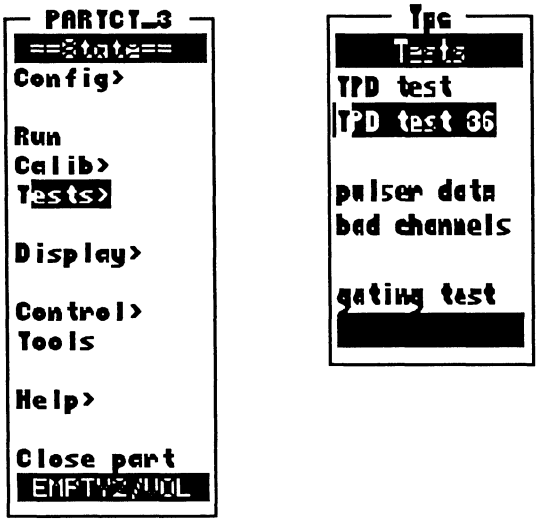

Fig 1: The partition controller- and Tests menus showing where TPD36 can be found.

The top menu of TPD36 can be seen in figure 2. It has the process name on the border, and the the name of the program as top title. All other UPI menus in the program have the menu name (i.e. the function of the menu) as top title.

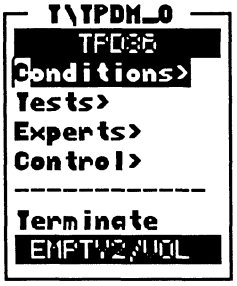

Fig 2: The top menu of TPD36.

2.2 Preparations for test

Before testing can begin some parameters have to be set. Go to the "Conditions" menu, figure 3.

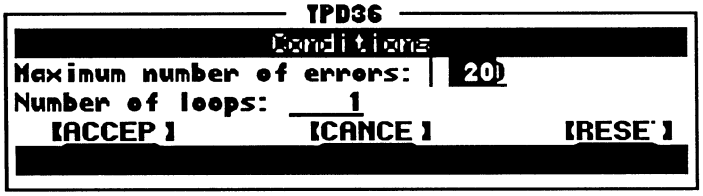

Fig 3: The "Conditions" menu where maximum number of errors and number of test loops are set.

On the "Conditions" menu the maximum number of errors and the number of test loops can be set.

The maximum number of errors determines when the program won't accept more errors. If it is exceeded in one of the tests, this test will be aborted and the remaining tests skipped. The default value is 20 but it can be set to any number between one and 200.

The number of loops determines how many times the selected tests are performed. All tests are executed once before the next loop starts. The default value is one, but any value between one and 1000 is accepted. Note that it takes about three minutes per loop to execute all tests on one sector.

The "Tests" menu is shown in figure 4.

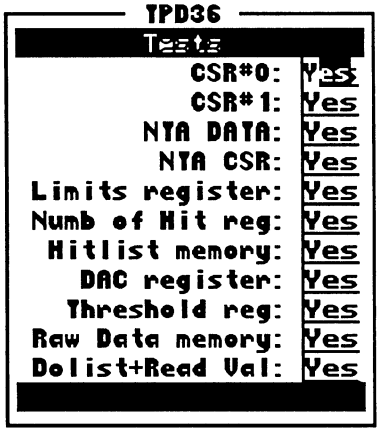

Fig 4: The "Tests" menu from which the desired tests are selected.

A certain test is selected by choosing "yes" on the line next to the test name on the "Tests" menu.

The first ten tests check the different registers in the TPDs. In general,for these tests, different test numbers (bit patterns) are written to the registers, read back and compared. The "Dolist+Read Val" test checks the DOLIST- and the Read Valid Data operations.

Here follows a brief description of what each test does:

- CSR#0: The TPD id '68210000'x is checked. The Hit Counter overflow error flag (bits<0>, <16>), SR enabling and SR assertion (bits<5:4>, <21:20>), DAQ enabling (bits<6>, <22>), TEST input enabling (bits<11>, <27>) and DAC writing enabling (bits<12>, <28>) are tested (the second bit mentioned at each function refers to the reset of the function).

- CSR#1: Tests the DAQ counter (which determines the bucket number during DAQ; bits<8:0>) and the two bank counters used during DOLIST (bits<17:16>) and DAQ (bits<19:18>).

- NTA DATA: Checks the high order address (bits<31:24>) to be '00'x (access to raw data memory) or '80'x (access to valid data register) and the access (address) counter (used during DOLIST; bits<14:0>).

- NTA CSR: Checks the NTA CSR register, that the high order address (bits<31:12>) is '00000'x (access to CSR#0 and CSR#1), '40000'x (access to Hitlist memory) or 'C0000'x (access to the Threshold-, Limits-,DAC- and Number Of Hits registers) and that the Hit counter (bits<10:0>) is OK.

- Limits register: Checks that the limits (bits<15:0>), i.e. the number of pre- and post samples and the minimum length of a pulse, can be set correctly.

- Numb of Hit reg: Checks the Number Of Hits register (which contains the number of hits to be read during Read Valid Data; bits<10:0>).

- Hitlist memory: Checks the Hitlist memory (bits<14:0>, <23:16>).

- DAC register: checks that the top- (bits<23:18>), linearity- (bits<17:6>) and the pedestal (bits<5:0>) values can be set and kept in the DAC register, which is used to load the values in the DACs. It does not check that these values are transferred correctly to the DACs.

- Threshold reg: Checks the Threshold register (bits<7:0>) (which is used as the data port to the threshold memory) and the threshold memory for each channel.

- Raw Data memory: Checks the Raw Data memory (bits<7:0>).

- Dolist+Read Val: Writes pulses in Raw Data memory and checks that the DOLIST operation, the number of hits and the Read Valid Data operation comes out correctly. This is done by comparing the results with software created Hitlist, Number Of Hits and Read Valid Data. In a way this is the ultimate test of a TPD since it uses all the different registers (except for the DAC register and most of the bits in CSR#0 and CSR#1) and also makes the TPD "active" (the DOLIST- and Read Valid Data

operations) in another way than the other tests.

The next line on the top menu says "Experts". This menu and its use will be explained in section 2.3.

Figure 5 shows the "Control" menu.

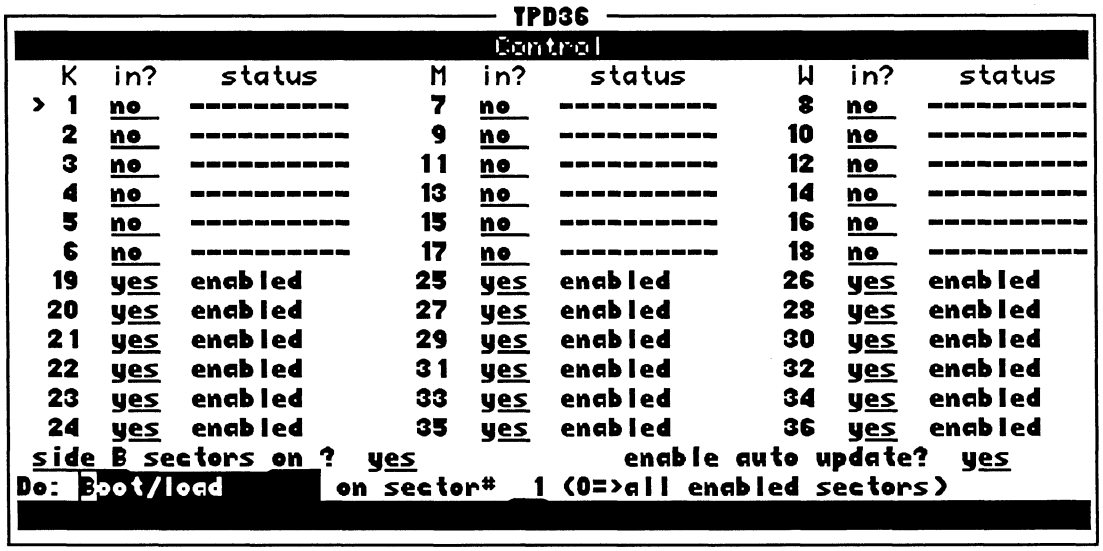

Fig 5: The "Control" menu, the main menu from which e.g. the tests are started.

The bigger part of the "Control" menu is a display of the sectors. Sectors that should be in the testing are selected by toggling to "yes" (which makes the status be "enabled") and sectors that should be out of the testing are excluded by toggling to "no" (which makes the status be "------") on the line next to each sector number.

The bottom line of the menu (from now on referred to as the action line) consists of an action ("Do:___") and a sector number on which the action should be performed ("on sector#__"). If this sector number is "0", the action will be performed on all sectors that are selected on the upper part of the menu. If its a number between one and 36, the action will be performed on this sector only. (Hence it's possible to have two or more sectors enabled while executing the action (e.g. "Boot/load") on one sector only.)

To select TPDs that should be tested, enter the sector number and toggle to the action "Setup TPDs" on the action line on the "Control" menu. The TPD setup menu appears, figure 6.

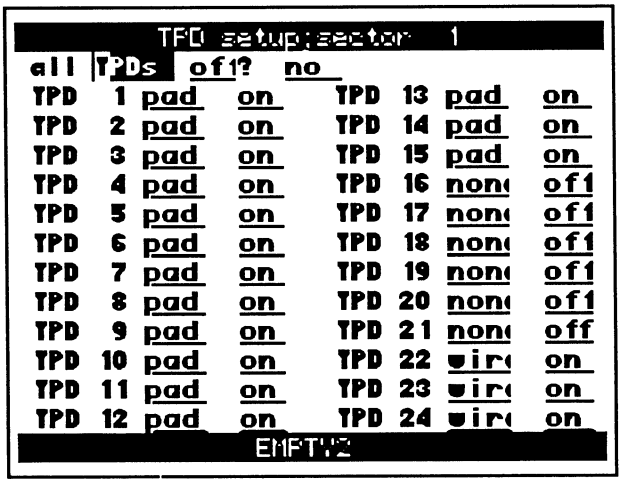

Fig 6: The "TPD setup" menu from which the TPDs to test can be selected.

The "TPD setup" menu shows the TPDs which are connected in the selected sector (according to the database on the VAX) and the TPDs which should be tested.

A TPD is connected when it's marked "pad" or "wire" and not connected when it's marked "none". The type of TPD is of no interest in TPD36 but is there since this is a "standard" menu, used also in the calibration- and related programs.

Each TPD is included or excluded in the tests by turning them "on" or "off" as seen in figure 6. However, this is meaningful only when testing a single sector and not when testing two or more sectors at the same time. When testing more than one sector, all the TPDs in all the selected sectors will be included.

2.3 Running and getting results

The "Control" menu is the menu from which the testing is controlled. All the necessary steps that needs to be taken to start the tests, look at the program status on the TPPs and upload the results are performed from the action line, at the bottom of the "Control" menu.

First the tests, i.e. the program running on the TPPs, need to be loaded in the TPPs. This can be done with the "Boot/load" or the "Multiboot/load" command on the action line. The former uses a loop in the program to do boot and load in all selected sectors and the latter uses the TPP state manager to boot and load TPPs simultaneously in the selected sectors. "Multiboot/load" saves a lot of time when many sectors are selected. If the booting fails in some TPPs when using this command, just redo "Multiboot/load" until all sectors have been booted and loaded - the TPP state manager will reboot only those sectors that failed to boot earlier. This is not the case with the "Boot/load" command which has to be reused

on one TPP after the other for those that failed to boot. The status of the program on each TPP is displayed on the upper part of the "Control" menu. Before booting and loading a TPP its status should be "enabled" (cf figure 5). A TPP has been booted and loaded all right when its status changes to "Booted".

After the tests have been loaded in memory in the TPPs they can be started with the "Start" command on the action line. The program has started when the status, after a few seconds, change to "Running" for the corresponding TPP. Check the "enable auto update __" option (explained below) if the status is not updated.

When the program has executed all tests in a sector, the status changes to "Done". The results can now be uploaded to the VAX. Enter the sector number (or 0 if results from many sectors are to be uploaded) on the action line ("on sector# __"; cf figure 5), toggle to the "Upload results" command and press return. The results from the sector(s) are now uploaded to the VAX and can be displayed on the "Results" menu. The status "Done E=#" on a sector status line indicates that the tests detected # number of errors in that sector. If testing several sectors and some are done before others, it's possible to upload the results from those sectors that are done while the tests finish on the rest.

There is a "Stop" that can be selected on the action line. This action will stop the program(s) on the TPP(s), but the status on the "Control" menu will not change accordingly.

One of the parameters on the "Control" menu reads "enable auto update __". This parameter is used to tell the program if automatic check and updating of the program status on the TPPs is to be performed every five seconds. Set to "yes" this will cause the status of the selected sectors to show the current state of the testing. This is achieved through an infinite loop in the program which can only be broken by an interrupt from the keyboard. If any command is pressed, the automatic updating will stop in order to give room for other activities, e.g. uploading of and looking at results for sectors that are already tested. The automatic updating can be resumed by choosing "Check status" on the action line. It starts automatically after "Start", "Boot/load" or "Multiboot/load" if auto update is enabled (set to "yes").

In this context the "Experts" menu should be mentioned which solves a somewhat intricate problem. When the program receives an interrupt from the keyboard it increments a counter. This counter keeps track of how many commands that are waiting to be performed and if it's greater than zero it makes the update loop stop. The counter is decremented for each command the program executes, until it reaches zero, which makes the program wait for input or start to loop again. However, due to obscure facts in UPI, this counter is incremented without reason occasionally (very rarely though). No command is executed on these occasions and the

counter is consequently not decremented. This makes it impossible to go into the update loop since the counter always will be greater than zero. The "Experts" menu is used to reset the counter to zero again.

2.4 Look at and interpret error messages

To look at the results for a sector that has been uploaded, choose "Display results" on the action line for the interesting sector. The "Results" menu appears, with the sector number in the top title, see fig 7.

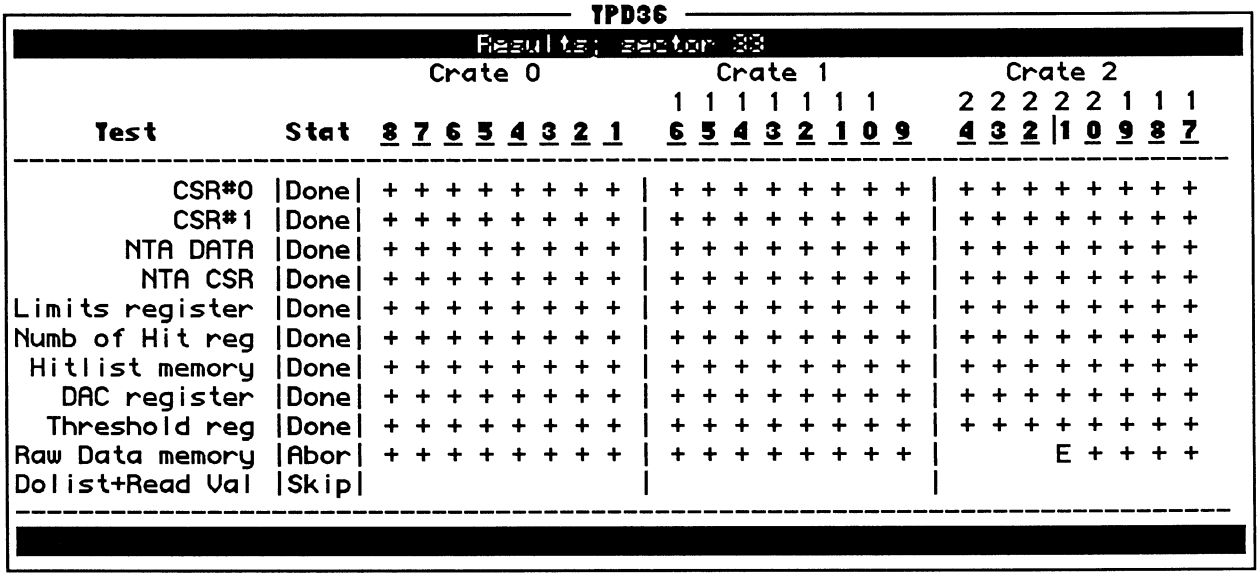

Fig 7: The "Results" menu on which the test results are displayed.

The outcome of the tests are displayed in the four rightmost parts i.e. in the "Stat" part and the three "Crate" parts. Each row corresponds to one test and each column in the three "Crate" parts corresponds to one TPD, numbered at the top of the menu.

The "Stat" (status of each test when it stopped) can be one of four values: "Done" if the test was completed, "Abor" if the test was aborted before it was completed, "Skip" if it was skipped (which happens if a previous test was aborted) and " " if the test was not selected.

In the three "Crate" parts of the menu, each TPD is marked with one of the following marks for each test: "+" if the test was successful, "E" if the test found at least one error, "." if the TPD was not connected or not selected and " " if the result is not known (e.g. when the test was skipped).

To get a more detailed error report for each TPD, move the cursor to the row with the TPD numbers at the top of the menu. Press return for the interesting TPD. An error report follows, see figure 8.

```
Sector: 33    TPD#: 21    FB addr: AE00000F    Date:  6-DEC-1989 10:05:40.81
Test          Error message                    Chn Bnk  Wrote    Read    Address
-------------------------------------------------------------------------------
RawMem:Read value is not correct               27  0      6C       0       D800
RawMem:Read value is not correct               27  0      6C       0       D801
RawMem:Read value is not correct               27  0      6C       0       D802
RawMem:Read value is not correct               27  0      6C       0       D803
RawMem:Read value is not correct               27  0      6C       0       D804
RawMem:Read value is not correct               27  0      6C       0       D805
RawMem:Read value is not correct               27  0      6C       0       D806
RawMem:Read value is not correct               27  0      6C       0       D807
RawMem:Read value is not correct               27  0      6C       0       D808
RawMem:Read value is not correct               27  0      6C       0       D809
RawMem:Read value is not correct               27  0      6C       0       D80A
RawMem:Read value is not correct               27  0      6C       0       D80B
RawMem:Read value is not correct               27  0      6C       0       D80C
```

Fig 8: Error report for one TPD.

The first line of the report header gives the sector- and TPD numbers, the FB address of the TPD and the date. The next line divides the report into seven columns: the test name to the very left, then a short description of the error and then some additional information about e.g what went wrong and where the error occurred in the TPD. The meanings of these five rightmost columns are as follows:

- "Chn" is the channel, numbered from 0 to 63, in which the error occurred.
- "Bnk" is the raw data bank, numbered from 0 to 3, in which the error occurred.
- "Wrote" is, if "Read" is not reported, what the test was writing to the specified location in the TPD when the error occurred (e.g. if there was a FB error during a write operation). If "Read" is reported, "Wrote" is the correct value that "Read" should be (e.g. if the test reads back a different value from what it wrote to a register or if the "Dolist+Read Val" test reads out an incorrect hitlist from the hitlist memory after the DOLIST operation).
- "Read" is the value that the test read from the specified location in the TPD.
- "Address" is the Fastbus address in the TPD where the error was detected. This is not always straight forward though. E.g. in the case of errors in the Threshold register, the "Address" will be 'C0000000'x which is the address to the CSR register used as a port to the Threshold memory, but the channel number to which the threshold should be written must first be loaded in NTA DATA.

The extra information given in these five columns differs between different tests and between different types of errors. The program tries to select the information that is of interest for each particular error and leaves the column blank if that specific information is without interest.

Press any key to go back to the "Results" menu and "Back Space" (or control-H) to go to the "Control" menu.

All detected errors can be written to a file and printed if "Setup output" and the sectors of interest are selected on the action line. This action leads to the "Output" menu, figure 9.

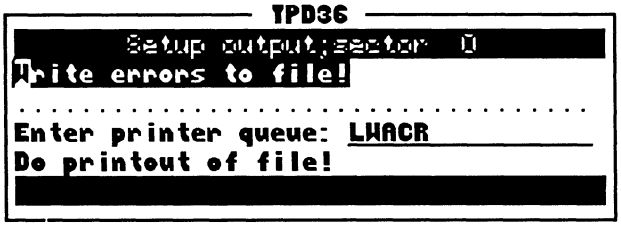

Fig 9: The "Output" menu, from which the error report can be written to a file and printed.

All commands on the menu refers to the sector number in the top title (which is the same number as on the action line). "Write errors to file!" makes the program look for TPDs with errors among the selected sectors and write the error report for those TPDs to a file ("[tpc.tpd_test.multi.result]tpd_err.txt"). "Do printout of file!" prints this file on the printer specified with the queue on the previous line "Enter printer queue:_____", see figure 9.

To quit TPD36, go to the top menu and select "Terminate".

3. SOFTWARE GUIDE

The intention with this part, the software guide, is to describe what is done in the test routines more elaborate than in section 2. It also gives some extra information that could be helpful for a person that wants to make changes to the program, e.g. add new tests or change the existing test routines. A basic understanding of Fastbus and a good understanding of the TPD modules is assumed throughout this section of the document.

3.1 TPP part

The files needed for the TPP part is found on network disk TPP_13, currently mounted as /n13 on the Aleph VAX online cluster, in the directory /VTT. Type "os_9> dir /n13/VTT" on a TPP workstation to have a look at it's content.

Source code to the TPP program is divided in three files: "tpd_test.f", "tests.f" and "dolist.f". In addition there are the include files "tpd_test.inc".and "dolist.inc". As a help to read the code, all variables in the TPP part are explained in appendix A.

3.1.1 Structure

The structure of the program is shown schematically in figure 10.

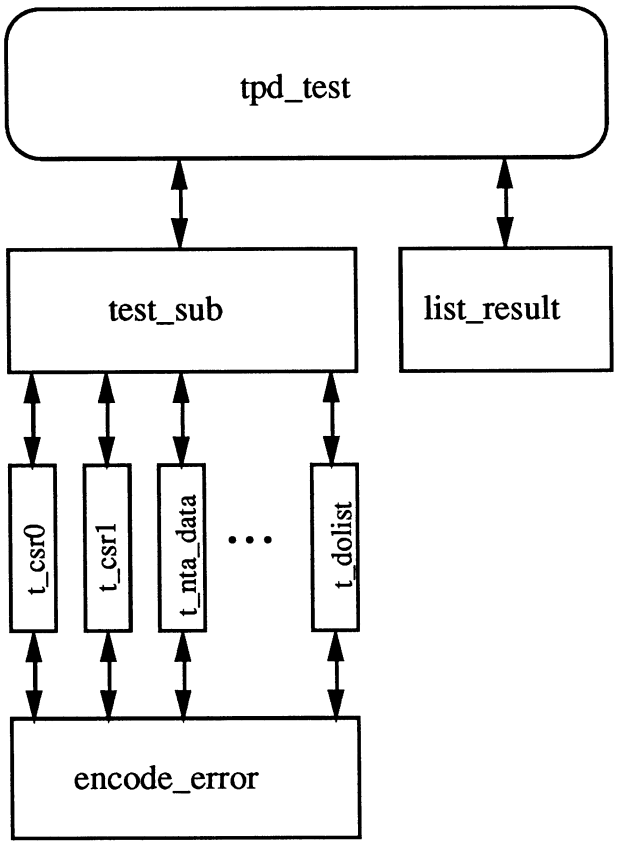

Fig 10: The structure of the TPP part of TPD36.

The top box is the main program (i.e. not a subroutine). This is the part that receives the test parameters from the VAX. (The communication with the VAX will be further explained in section 3.2.3.).

Once the parameters and the start-of-run signal have been received on the TPP, the main program calls "test_sub". This subroutine loops over all the test subroutines a certain number of times ("loops") given by the user. If an error is detected by one of the test routines, it will call "encode_error". This routine encodes the error and puts it in the array "result" that can be uploaded to the VAX. When all the tests are done, the control is returned to the main program which calls "list_result". "List_result" is an error log routine which is useful during debugging periods, e.g. when new tests are being installed. It decodes the detected errors in a similar way as on the the VAX and writes the results to a remote terminal, i.e. a terminal connected via ethernet. When developing new tests however, it's easiest to work with the TPP only, see section 3.3.

3.1.2 Test routines

As mentioned in section 2.2, the first ten tests are similar in structure and function. It's enough to describe one of these tests in detail to get an idea of how the rest of the register- and memory tests are built up as well. The CSR#0 register test has been chosen as an example. Its test procedure is analysed in section 3.1.2.1. This section also makes clear the use of the

non test routines in the TPP part of TPD36. For the other register- and memory tests it's only summarised what they do. Lastly, the DOLIST test is described.

3.1.2.1 Example of a register test: CSR#0

Look in the "t_csr0" subroutine in the file "/n13/VTT/tests.f".

As all test routines the CSR#0 test begins with a check if the maximum number of errors ("max_err") which was set on the VAX has been exceeded. If not, the test starts to loop over all TPDs in the sector.

From now on, all that is done is a number of Fastbus write- and read operations [1] to the CSR#0 register. The test first reads the register and checks that bits <31:16> (the module id) is '6821'x, and after that reset the TPD by setting bits CSR#0<31:30>. It now starts to test all the bits that are used in the register, except for bit <8> since this bit enables the DOLIST operation and is anyway tested in the DOLIST test. Every bit corresponds to a flag that can be set and reset and that is what the CSR#0 test does: it sets one of the flags, checks that it has been set, resets the flag and checks that it's gone, sets the next flag and so on (cf [2] to see the exact use of the CSR#0- and the other registers). All the numbers that are written to set and reset the flags are defined in the beginning of the test routine in the array "test_arr(12)". If an error is detected during a Fastbus operation or in a comparison between what was written to and read from the register, there is a call to the subroutine "encode_error" which encodes the error and puts the result in the array "result".

The other nine register/memory tests do not necessarily test the bits one by one. In some tests, different numbers, "bit patterns", are written, read back and checked. These bit patterns are created in the routine "get_pattern" at the end of "tests.f".

When the loop over all TPDs is completed, the "test_status(12)" array is updated. This array determines the status ("Stat") of the test on the "Tests/results" menu on the VAX and its numbers are interpreted as follows: 0=>' ', 1=>'Done', 2=>'Abor' and 3=>'Skip'.

Information for the "status" line on the "Control" menu (figure 5) on the VAX is managed by keeping fresh status information in the CSR#11 register in the TPP. This register is read from the VAX via Fastbus every five seconds (when "enable auto update" is set to "yes") and its bit allocation is: number of currently detected errors (bits <23:16>), the number of the last done test (bits <11:8>) and the program status (bits <1:0>; 0=>"Unknown", 1=>"Booted", 2=>"Running" and 3=>"Done"). The CSR#11 register is updated by a call to "update_state" and this is done before all tests start (to update the program status), every time an error is detected (to update the number of errors), at the end of each test (to update the number of the last done test) and when all tests have been

executed (to update the program status). Hence, the last action in "t_csr0" is a call to "update_state" which reports, to the CSR#11 register, that the CSR#0 test is done.

Summary of the CSR#0 test:

- Bits <31:16> (the module id) is read out and checked to be '6821'x.
- The TPD is reset by setting bits CSR#0<31:30> (reset module including CSR and reset registers).
- Bit patterns ('00000001'x, '00000010'x, '00000020'x, '00000040'x, '00000080'x and '00001000'x) are written to bits <0> (set error flag), <4> (enable SR assertion), <5> (assert SR), <6> (enable DAQ), <11> (enable test input) and <12> (enable DAC writing). The patterns are read back and compared and finally reset by writing other bit patterns ('00010000'x, '00100000'x, '00200000'x, '00400000'x, '08000000'x and '10000000'x) to bits <16> (clear error flag), <20> (disable SR assertion), <21> (reset SR), <22> (disable DAQ), <27> (disable test input) and <28> (disable DAC serial load). Lastly, the patterns are checked to be reset. Each pattern is written, read and reset before the next one.
- The whole TPD is reset by setting bits CSR#0<31:30>.

3.1.2.2 The CSR#1 register test

The test checks the CSR#1 register:

- Bit patterns ('00000000'x, '000F01FF'x, '00050155'x, '000A00AA'x, sliding 0 and sliding 1) are written, read back and compared for bits <19:16> (bank number definition during DAQ and DOLIST).

3.1.2.3 The NTA DATA register test

The test checks the NTA DATA register:

- The TPD is reset by setting bits CSR#0<31:30>.
- Bits <31:24> (high order address) are read out and checked to be '00'x (access to raw data memory) or '80'x (access to data space where valid data is read out).
- Bit <15> (not used) is checked to be 0.
- Bit patterns ('00000000'x, '0007FFF'x, '00005555'x, '00002AAA'x, sliding 0 and sliding 1) are written, read back and compared for bits <14:0> (access address counter for raw data memory)

3.1.2.4 The NTA CSR register test

The test checks the NTA CSR register:

- The TPD is reset by setting bits CSR#0<31:30>.
- Bits <31:11> (high order address) are read out and checked to be '00000'x (access to CSR#0 and CSR#1), '40000'x (access to hitlist

memory) or 'C0000'x (access to threshold-, limits-, DAC- and number of hits registers).
- Bit patterns ('40000000'x, '400007FF'x, '40000555'x, '400002AA'x, sliding 0 and sliding 1) are written, read back and compared for bits <10:00> (hit counter).

3.1.2.5 The Limits register test

The test checks CSR#$C0000001, i.e. the Limits register:

- Bit patterns ('00000000'x, '0000FFFF'x, '00005555'x, '0000AAAA'x, sliding 0 and sliding 1) are written, read back and compared for bits <15:00> (space where the limits are set).

3.1.2.6 The Number of Hits register test

The test checks CSR#$C0000003, i.e. the Number of Hits register:

- Bit patterns ('00000000'x, '000007FF'x, '00000555'x, '000002AA'x, sliding 0 and sliding 1) are written, read back and compared for bits <10:00> (space where the number of hits can be read).

3.1.2.7 The Hitlist memory test

The test checks CSR#$40000000 to $400007FF, i.e. the Hitlist memory:

- The TPD is reset by setting bits CSR#0<31:30>.
- Bit patterns ('00000000'x, '00FFFFFF'x, '00555555'x, '00AAAAAA'x, sliding 0 and sliding 1) are used to create data blocks of 2000 identical longwords. The blocks are written and read back in block transfers and compared for bits <23:00> (space where the hitlist is stored).

3.1.2.8 The DAC register test

The test checks CSR#$C0000002, i.e. the DAC register:

- The serialization circuit is enabled by setting bit CSR#0<12>.
- Channels are selected by writing channel number in NTA DATA<14:9> (channel definition bits in the address access counter. All channels are tested). Bit patterns ('00000000'x, '00FFFFFF'x, '00555555'x, '00AAAAAA'x, sliding 0 and sliding 1) are written to bits <23:00> (space where values of the DACs are set). The program waits until serialization is done by checking CSR#0<13> (status DAC serial load). The patterns are read back and compared.

3.1.2.9 The Threshold register test

The test checks CSR#$C0000000, i.e the Threshold register:

- Channel is selected by writing channel number in NTA DATA<14:9> (channel definition bits in the address access counter. All channels are tested).
- Writes the channel number as threshold value to bits <7:0> (space where threshold is set).
- Channel is selected by writing channel number to NTA DATA<14:9>.
- Values are read back and compared with what was written.
- Channel is selected by writing channel number to NTA DATA<14:9>. Bit patterns ('00000000'x, '000000FF'x, '00000055'x and '000000AA'x) are written, read back and compared for bits <7:0>.

3.1.2.10 The Raw Data memory test

The test checks the Raw data memory:

- The TPD is reset by setting bits CSR#0<31:30>.
- Channels are selected by writing channel number in NTA DATA<14:9> (channel definition bits in the address access counter. All channels are tested) and banks by writing bank number in CSR#1<17:16> (space where bank is defined during DOLIST).
- 256 different (values between 0 and 255) data blocks of 512 identical bytes are created and written in block transfers, block 1 to channel 0,bank 0, block 2 to channel 0, bank 1 etc.
- Channels are selected by writing channel number to NTA DATA<14:9> and banks by writing bank number in CSR#1<17:16>.
- Values are read back from all banks in block transfers and compared with what was written.

3.1.2.11 The DOLIST test

Look in the file "/n13/VTT/dolist.f" which contains all subroutines related to the DOLIST test.

The DOLIST test is just a little bit more complicated than the others. There are two phases in the test. The first phase makes the initialization. This is to define the limits and thresholds to be used during the test, to create raw data pulses that will be written to the Raw Data memory, to create a software hitlist, to calculate the number of hits and to pick out the valid data. The second phase is the actual testing, which loops over the TPDs, writes limits and raw data, starts the DOLIST operation, reads out hitlist, hitnumber and valid data and compares the results with corresponding results created in the program.

The test starts off with a check that the maximum number of errors ("max_err") is not exceeded. Then the limits and threshold of the pulses (to be written to the Limits- and the Threshold registers respectively) are defined ("pre", "post", "length" and "threshold").

The next step is to create the pulses to be written to the Raw Data

memory which is done in "dol_create_pulse". This routine creates one triangular pulse for one channel (the same set of pulses will be used for all four banks in a channel) according to the specifications given in the arguments. For each call to "dol_create_pulse", a new pulse is added. Hence, a pattern of several pulses is created by calling this routine several times for each channel. In the present version of the test, parts of the pattern is shifted a few buckets for each channel by letting the middle bucket (i.e. the center of the pulse) of some pulses vary with the channel number. The result is stored in the array "raw_data(0:63,0:511)".

The software hitlist ("softlist(2048)") is created in "dol_create_softlist" which also counts the number of hits, the number of valid samples and picks out the valid data ("soft_valid_data(max_valid)") from the array "raw_data". The number of hits and the number of valid samples are stored in "hit_nb" and "valid_samp". Note that number of hits and number of valid samples for TPDs that contains less than 64 channels are stored at the end of "dol_create_softlist" in separate variables. This is necessary to avoid creating the raw data and hitlist separately for these TPDs.

The "dol_" routines have been written in a way to make it easy to add new pulses or to change the shape of the pulses: just add calls to "dol_create_pulse" with the new characteristics (described in the header of "dol_create_pulse"). There is one important thing to note however: if the width of any pulse, the number of pulses or any of the variables "pre" or "post" are changed the parameter "max_valid" HAS TO BE updated as follows:
max_valid=64*(width#1+width#2+.....+{number of pulses}*(pre+post)).
"max_valid" determines the size of the arrays "valid_data" and "soft_valid_data" and has been introduced to make their sizes as small as possible in order to save memory space in the TPPs.

The rest of the test is actually just a lot of Fastbus write- and read operations to/from different registers in the TPDs and their purposes should be transparent when looking in the code. One remark though: right after the DOLIST operation has been started, there is a call to the RTF runtime routine "f_sleep(5,0)". This call is necessary to make the program hibernate in 0.05 seconds while the DOLIST operation finishes (cf [2] for an explanation of the DOLIST execution time - 0.05 sec is actually quite a bit longer than the DOLIST operation takes).

Summary of the DOLIST test:

- Raw data is created for one bank and all channels.
- Software hitlist and valid data is created from the raw data. Number of hits and number of valid samples is calculated.
- The TPD is reset by setting bits CSR#0<31:30>.
- Limits are set in the Limits register <15:00> (presamples=2, postsamples=2 and length=6).
- Bank is selected in CSR#1<17:16>, channel in NTA DATA<14:9>,

thresholds are defined in the Threshold register <7:0> (threshold=6) and raw data are written to Raw data memory for all channels. (Not used channels are "turned off" by setting threshold=255).
- The Dolist operation is started by setting bit CSR#0<8> and the test waits 0.05 seconds until the Dolist operation is done.
- The Dolist status bit CSR#0<8> is checked to be 0 (dolist is done).
- The hitcounter NTA CSR<10:00> is read out and compared with the calculated number of hits.
- The hitlist memory is read out in block transfer from CSR#$40000000 and compared with the software hitlist.
- Address to the first pulse to be read during Read Valid data is loaded in NTA CSR<31:00>, the number of hits to be read is loaded in Number of Hits register, a block transfer of the valid information is done from DATA space address $80000000 and compared with software extracted valid data.

3.1.3 How to link and compile

The link step, to make "tpd_test" (the TPP part of TPD36), should be made in the following way: "l68 -a /r0/LIB/rtfstart.r tpd_test.r tests.r dolist.r r0/LIB/sys.l /n6/PUL/pullib.r /n6/VTT/vtt_comm.r /n7/RTF/LIB/tpp_comics.l /n7/RTF/RELS/fbpack.r /n7/RTF/RELS/tpclib.r /n7/RTF/RELS/mac_lib.r /n7/RTF/RELS/tpp_syslib.r -o=tpd_test".

There is a makefile in /n13/VTT that can be used to compile and link the TPP part of TPD36 and to load the necessary files for an editing session to the current work directory on the RAM disk on the TPP (cf [3] for an explanation of the make utility and "makefile"). An editing session could proceed e.g. like this:
Log in to a TPP workstation and create a work directory on the RAM disk. Copy the makefile from the /n13/vtt library ("os_9> copy /n13/VTT/makefile"). Use the makefile to copy all the necessary files from /n13/VTT. Type: "os_9> make load". This will copy all the files of interest to the current work directory. Do the editing and copy the edited file back to /n13/vtt ("os_9> copy tests.f /n13/vtt/tests.f -r"). This is important since the makefile searches /n13/VTT and not the workdirectory for files that has been updated. Type "os_9> make tpd_test" and the executable image "tpd_test" will be created. If the compiler complains "file not found", make sure that all files that are related to the RTF compiler are loaded in memory (e.g. "rtflib and" "rtfdat"). Copy "tpd_test" and the new object files back to /n13/VTT by typing "os_9> make store".

Before the TPP part can be used in TPD36 it has to be transferred to the execution directory on the VAX and transformed into an ".ebf" file that can be downloaded to the TPP. These two steps are easily done with the "ebf" command on the VAX. Go to the execution directory "disk$tpc:[tpc.tpd_test.multi.nodeb]" and type "$ ebf /tpp_13/vtt/tpd_test". The ".ebf" file is created with the name

"tpd_test.ebf".

3.2 VAX part

This section is not intended to give a complete description of the VAX part of TPD36. There are however some comments to be made, e.g. about the VAX-TPP communication and the menu environment. This section is also here to make it possible to get a complete view of the program.

The VAX part files are located in disk$tpc:[tpc.tpd_test.multi.source] (tpd36.for and tpd36.inc) and in disk$tpc:[tpc.tpd_test.multi.nodeb] (descrip.mms and tpd36.exe).

All variables are explained in appendix B. No pure naming convention has been used, but variables and routines have got names that indicates what they do and variables which are similar have been given similar names (e.g. channel- and bank number are named "chan_nb" and "bank_nb" respectively).

3.2.1 Structure

Since the program runs under the switcher, it uses the core skeleton [4]. The core skeleton gives the basic structure to the program. A schematic picture of the structure, or flow, of the VAX part of TPD36 is shown in figure 11.

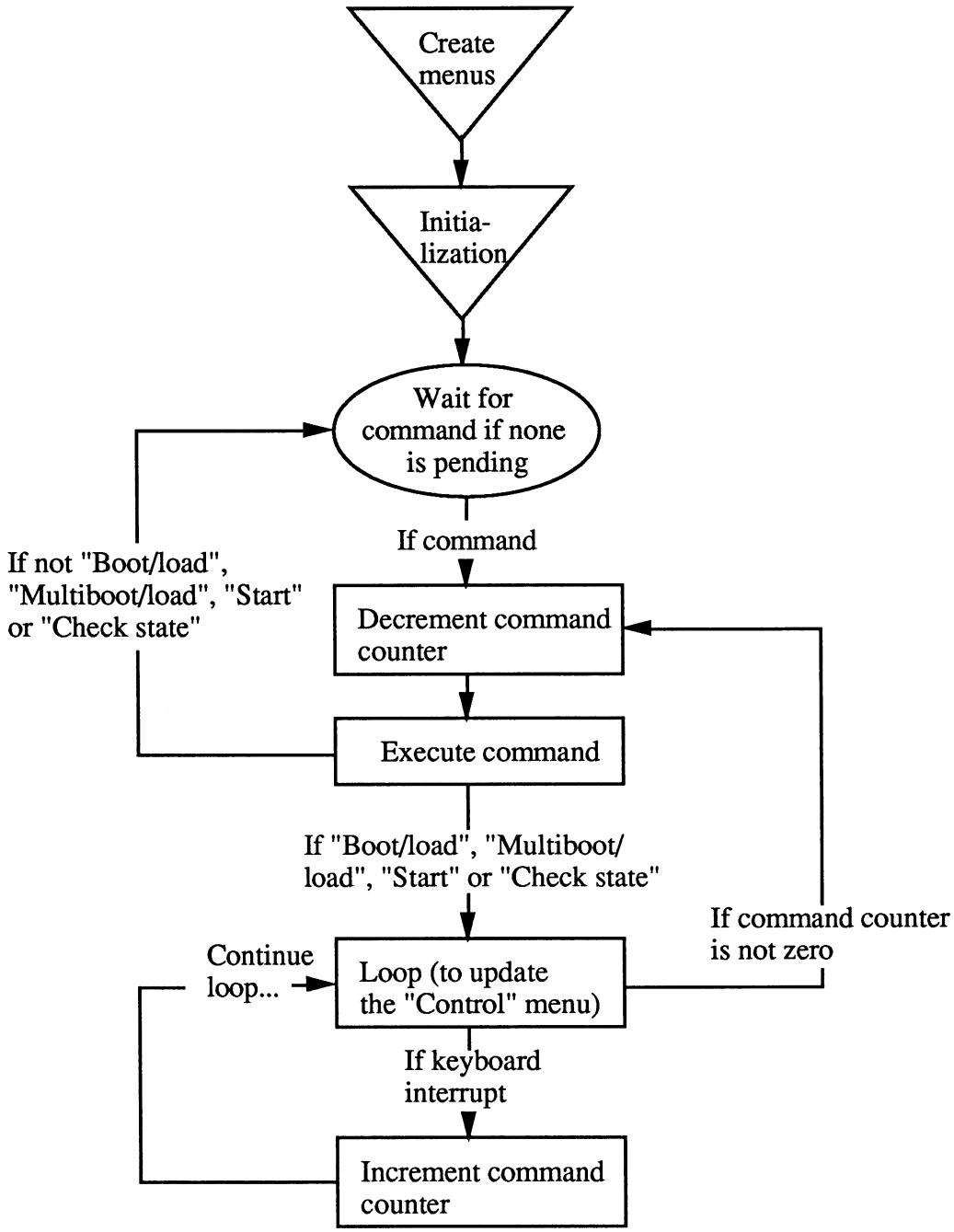

Fig 11: Structure of the VAX part of TPD36.

Initialization is done in "pulser_user_creation". This routine is called from the normal core routine "user_creation" which is provided by the pulser_user package [5].

There's a straight forward correspondence between each menu command and a certain set of operations and routine-calls grouped together in the routine "user_input". The depth of calls seen from this routine is actually not deeper than two. Look in the header of each routine to get a description of what it does.

At the end of "user_input" the program starts an infinite loop if the commands "Boot/load", "Multiboot/load", "Start" or "Check state" have

been issued on the "Control" menu. The loop stops only if the command counter "user_interrupt" is incremented. This is done in the routine "user_receive_ast" which is called as soon as any command on any menu is pressed. It's possible that the program receives two or more interrupts (commands) before it starts to execute the first. For each command that is executed, the command counter is decremented until it reaches zero and the update loop can be entered again.

3.2.2 Menu environment (pulser_user library)

TPD36 resembles the calibration and related programs in some aspects. The programs runs on the TPPs in all sectors and makes use of the TPDs. All these programs uses the same "Control" menu and "TPD setup" menu in order to create a somewhat standardized interface. The routines that sets up the menus are provided in the "pulser_user" package [5].

Note however: the "Control" menu needs the infinite loop to update the status of the sectors. This loop is only broken when the program receives an asynchronous interrupt from the keyboard. But this can only be accomplished if the program runs under the switcher. The program will not work properly if started stand alone.

3.2.3 Communication with TPP (vtt library (tpplib))

To manage the communication between VAX and TPP, TPD36 uses the VTT library [5] (a disguised form of tpplib [6]) and the TPP state manager [7]. The VTT library is used for all communication (boot, loading of program, downloading of test variables etc) except for the parallell boot and program loading which is done with the TPP state manager. Both these libraries, requires that the program is an ".ebf"-file. The procedure to create this file was described in section 3.1.4.

3.2.4 How to link and compile

There is an mms file located in "disk$tpc:[tpc.tpd_test.multi.nodeb]" that can be used to link and compile the VAX part of TPD36. The use of each link step is explained in the file.

3.3 Development of TPP part (TPP stand alone version)

It's a very tedious work to develop new tests if the test module has to be transformed into an ".ebf"-file after each small change to the code and started from the VAX to be tried out. For this reason all tests can be developed in a TPP stand alone version of the test program. The program is not userfriendly (it's not supposed to be used by ordinary users) but the structure is very similar to the TPP part of TPD36 and is easily understood if the two are compared.

The files can be found on network disk TPP_13 (currently /n13) in the

directory /n13/TPP. The files are the same as for the TPP part of the VAX/TPP version (but with the prefix "tpp_" on the names) including a makefile for the compile- and link procedure.

References

[1] P.S. Marrocchesi, ALEPH TPP FastBus Library, version 1.0 March CERN 1989
[2] B. Lofstedt, T.P.D.-Time Projection Digitizer, CERN-EP 1989.04.13
[3] Using Professional OS-9 Version 2.1, Microware Systems Corporation June 1987
[4] J. Bourotte et al, The CORE Template Program, ALEPH 89-95 DATACQ 89-12 5 CERN June 1989
[5] J Conway et al,The ALEPH TPC Electronics Calibration/Test Pulser System, CERN June 1989
[6] P.S. Marrocchesi, TPPLIB, ALEPH 89-109 TPCGEN 89-12 CERN 28.6.1989
[7] B. LeClaire, The TPP state manager, TPC-50 CERN September 1989

Appendix A

Global variables used in TPP part of TPD36

name: bufmode
type: INTEGER*4
description:

name: chan_nb
type: INTEGER*4
description: The current channel number in the tests.

name: eid
type: INTEGER*4
description: dummy variable that is put at the environment id's place in the FB routines

name: err_cntr
type: INTEGER*4
description: error counter on the TPP (to VAX)

name: hit_nb
type: INTEGER*4
description: software calculated number of hits.

name: hit_nb 15
type: INTEGER*4
description: software calculated number of hits for TPDs with last channel number 15.

name: hit_nb_31
type: INTEGER*4
description: software calculated number of hits for TPDs with last channel number 31.

name: hit_nb_47
type: INTEGER*4
description: software calculated number of hits for TPDs with last channel number 47.

name: hit_nb_63
type: INTEGER*4
description: software calculated number of hits for TPDs with last channel number 63.

name: ibits
type: INTEGER*4
description: FUNCTION that picks out a given number of bits from one argument and puts it in ibits (works the same way as the standard function in VAX Fortran).

name: inbuf
type: INTEGER*4
description: write buffer in the FB routines

name: last_chan
type: INTEGER*4

description: FUNCTION that returns the number of the last channel for the TPD given as argument

name: length
type: INTEGER*4
description: minimum length of pulse used during dolist.

name: loop_nb
type: INTEGER*4
description: Counts the test loops.

name: loops
type: INTEGER*4
description: number of test loops (from VAX)

name: max_chan=64
type: INTEGER*4
description: maximum number of channels in a TPD

name: max_err
type: INTEGER*4
description: maximum allowed errors (from VAX)

name: max_tests=12
type: INTEGER*4
description: maximum number of tests currently in the source code

name: max_tpd=24
type: INTEGER*4
description: maximum number of TPDs in a sector

name: max_valid
type: INTEGER*4
description: PARAMETER wich determines the size of "soft_valid_data" and "valid_data".

name: outbuf
type: INTEGER*4
description: read buffer in the FB routines

name: pad
type: INTEGER*4
description:

name: post
type: INTEGER*4
description: number of post samples to be used during dolist.

name: pre
type: INTEGER*4
description: number of pre samples to be used during dolist.

name: raw_data(0:511,0:63)
type: INTEGER*1
description: buffer with software created raw data for one bank and 64 channels.

name: result(200,8)
type: INTEGER*4
description: test results in encoded form (to VAX)

name: sect_nb
type: INTEGER*4
description: current sector number (from VAX)

name: softlist(2048)
type: INTEGER*4
description: Contains the software created hitlist.

name: soft_valid_data(max_valid)
type: INTEGER*1
description: Contains the software created valid data.

name: status(10)
type: INTEGER*4
description: array with the return code from the FB operations

name: test_status(max_test)
type: INTEGER*4
description: status of tests in encoded form when test finished (to VAX)

name: tests(max_tests)
type: INTEGER*4
description: the tests to be executed (from VAX)

name: tpd_pads(max_tpd)
type: INTEGER*4
description: primary addresses to the TPDs (from VAX)

name: threshold
type: INTEGER*4
description: raw data pulse threshold to be used during dolist.

name: valid_data(max_valid)
type: INTEGER*4
description: buffer in which valid data from the TPD are put.

name: valid_samp
type: INTEGER*4
description: software calculated number of valid samples.

name: valid_samp_15
type: INTEGER*4
description: software calculated number of valid samples for TPDs with last channel number 15.

name: valid_samp_31
type: INTEGER*4
description: software calculated number of valid samples for TPDs with last channel number 31.

name: valid_samp_47
type: INTEGER*4

description: software calculated number of valid samples for TPDs with last channel number 47.

name: valid_samp_63
type: INTEGER*4
description: software calculated number of valid samples for TPDs with last channel number 63.

Appendix B

Global variables used in VAX part of TPD36

name: action
type: CHARACTER*15
description: The action to be performed according to the action line on the "Control" menu.

name: action_list(10)
type: CHARACTER*15
description: List of possible actions on the action line on the "Control" menu.

name: all_err_cntr(36)
type: INTEGER*4
description: Array that keeps "err_cntr" from all sectors after upload from TPP.

name: all_result(36,200,8)
type: INTEGER*4
description: Array that keeps "result(200,8)" from all sectors after upload from TPP.

name: all_test_status(36,12)
type: INTEGER*4
description: Array that keeps "test_status(12)" from all sectors after upload from TPP.

name: comment1
type: CHARACTER*80
description: Comment line on the "Results" that writes out "Crate 0 .....etc".

name: cond_menu
type: INTEGER*4
description: Menu id for the "Conditions" menu.

name: contr_act
type: INTEGER*4
description: Line id for the action line on the "Control" menu.

name: contr_menu
type: INTEGER*4
description: Menu id for the "Control" menu.

name: contr_stop
type: INTEGER*4
description: Line id for stop command on the "Control" menu (currently not used).

name: csr10
type: INTEGER*4
description: Receives the content of CSR#10 in the TPP after each call to "vtt_get_csr".

name: csr11
type: INTEGER*4
description: Receives the content of CSR#11 in the TPP after each call to "vtt_get_csr".

name: err_cntr
type: INTEGER*4
description: The detected number of errors in one sector (uploaded from TPP).

name: exp_menu
type: INTEGER*4
description: Menu id for the "Experts" menu.

name: line
type: CHARACTER*80
description: Used to write the dashed lines on the "Results" menu.

name: loops
type: INTEGER*4
description: Number of times the program should loop over each test on the TPP (downloaded to TPP).

name: lun
type: INTEGER*4
description: Logical unit number used for the file where the error log can be written.

name: max_chan
type: INTEGER*4
description: PARAMETER =64 that determines the maximum number of channels in a TPD.

name: max_err
type: INTEGER*4
description: Maximum number of errors that are allowed (downloaded to TPP).

name: max_tests
type: INTEGER*4
description: PARAMETER=12 that determines the number of tests the program is currently prepared for.

name: option(max_tests)
type: CHARACTER*3
description: Contains 'Yes' or 'No ' according to the selection on the "Tests" menu (and the "Results" menu).

name: out_comm1
type: INTEGER*4
description: Comment line id for dotted line on the "Output" menu.

name: out_file
type: INTEGER*4
description: Command line id for the "Write errors to file" command on the "Output" menu.

name: out_menu
type: INTEGER*4
description: Menu id for the "Output" menu.

name: out_print

type: INTEGER*4
description: Command line id for the "Do printout.." command on the "Output" menu.

name: out_que
type: INTEGER*4
description: Parameter line id for the printer queue parameter line on the "Output" menu.

name: pr_flag
type: LOGICAL
description: Flag used in the error log routine to determine if the error log is written to file (.true.) or to the screen (.false.).

name: que_list(5)
type: CHARACTER*15
description: List of common printer queues that can be toggled on the "Output" menu.

name: que_name
type: CHARACTER*15
description: Printer queue name (selected on the "Output" menu) where error log is printed.

name: res_menu
type: INTEGER*4
description: Menu id for the "Results" menu.

name: result(200,8)
type: INTEGER*4
description: The detected errors from one sector in encoded form.

name: sect_com
type: INTEGER*4
description: Sector number that is selected after the action line on the "Control" menu (on which the action should be performed).

name: sect_nb
type: INTEGER*4
description: Sector number.

name: sect_upload(36)
type: INTEGER*4
description: Array of flags that contains the sector number if that sector has been uploaded and zero otherwise.

name: status(12)
type: CHARACTER*48
description: Contains the lines for each test on the "Results" menu to which the result is written with 'E', '+' or ' '.

name: status_list(max_tests)
type: CHARACTER*4
description: List of every possible status that could appear on the "Results" menu.

name: test(max_tests)
type: CHARACTER*10
description: Test names that appear on the "Tests"- and the "results" menus.

name: test_menu
type: INTEGER*4
description: Menu id for the "Tests" menu.

name: test_status(max_tests)
type: INTEGER*4
description: The status of each test in a sector in encoded form (uploaded from TPP).

name: tests(max_tests)
type: INTEGER*4
description: Array of flags that marks out wether a test should be included or not. Each position corresponds to one test. '1' means include test and '0' means exclude test. (downloaded to TPP).

name: text
type: CHARACTER*80
description: String used as a buffer when writing messages UPI message window.

name: top_comm1
type: INTEGER*4
description: Comment line id on the top menu.

name: top_cond
type: INTEGER*4
description: Command line id for command on the top menu leading to the "Conditions" menu.

name: top_contr
type: INTEGER*4
description: Command line id for command on the top menu leading to the "Control" menu.

name: top_exp
type: INTEGER*4
description: Command line id for command on the top menu leading to the "Experts" menu.

name: top_menu
type: INTEGER*4
description: Menu id for the top menu.

name: top_term
type: INTEGER*4
description: Command line id for command on the top menu to terminate the program.

name: top_test
type: INTEGER*4
description: Command line id for command on the top menu leading to the "Tests" menu.

name: tpd_menu
type: INTEGER*4
description: Menu id for the "TPD setup" menu.

name: tpd_pads(max_tpd)
type: INTEGER*4
description: Contains the primary addresses to the TPDs to be included in the tests and zero otherwise (downloaded to TPP).

name: tpp_output
type: CHARACTER*16
description: Used by the vtt library in the common block "tppout" to determine where output from the TPP will be written (' >>>/rt & ' writes TPP output to a remote terminal).

name: user_interrupt
type: INTEGER*4
description: Command counter used to interrupt the update loop and count down pending commands to zero.