PCLTS

LTS-2020 Test System Accelerator

User's Guide

Rev 1.13

Contents

Overview 1-1
The PCLTS accelerator for the LTS-2020 test system 1-1
ADVANTAGES OVER STANDARD LTS-2020 1-1
Production Test Floor impact 1-1
Engineering Impact 1-2
How the increase in speed is achieved 1-3
Conversion of Existing Basic Programs 1-3
Hardware Installation 2-1
System Requirements 2-1
Installing the CPU Emulator Board 2-1
Installing the Bus Driver Board 2-3
Running the Diagnostics Program 2-4
Setting Up the Printer 2-4
Setting the Printer's Character Set 2-4
The Printer Port 2-5
Parallel Printers 2-5
Serial Printers 2-5
Software Installation 3-1
Software Requirements 3-1
Installing Microsoft Visual C++ Compiler 3-1
Microsoft Visual C++ V1.0 and V1.5 3-1
Microsoft Visual C++ V2.0 3-2
Microsoft Visual C++ V1.52 and Visual C++ V4.00 3-3
A Couple of Important Details 3-3
Installing the PCLTS System Software 3-4
The PCLTS Directories Tree 3-4
The LTSSHEL.INI Configuration File 3-5
Maximizing Available DOS Memory 3-7
386MAX Memory Manager 3-7
Quarterdeck's QEMM 3-8
Creating a Test Program 4-1
The Test Program 4-1
Creation of a Test Program 4-1
Recording the Test's Parameters and Results 4-2
tnum is Incremented Automatically 4-2
Binning 4-3
Differences with LTS Basic 4-3
Setting the Fail Bin 4-3
Disqualifying Passing Bins When Down-grading 4-3
The test_downgraded variable 4-4
Down-grading Example 4-4
Function Bodies Required in the Test Program 4-4
test_program() function 4-5
define_bin_descriptions() function 4-5
user_before_testing_part() and 4-6
user_after_testing_part() functions 4-6
Other Required Include Files and Definitions 4-6
Include Files 4-6
Variable Declarations and Function Prototypes 4-7
Using a Limit Table 4-7

Loading the Limit Table 4-8
File Naming Convention 4-8
Limit File Format 4-8
Table Rows 4-9
Table Columns 4-10
Pass Bin 4-10
Beginning and Ending Test Numbers 4-10
Test Name 4-10
Units 4-10
Formatting Decimals 4-10
Fail Bin 4-11
Limit Pair Columns 4-11
An Example of a Simple Test Program 4-11
Same Program Without a Limit Table 4-15
Compiling the Test Program 4-15
Linking 4-16
Debugging the Test Program 5-1
Using Microsoft's Codeview Debugger 5-1
Opening up the Necessary Windows 5-1
Setting a Trap at the Start of test_program() 5-2
Running the Program 5-3
Trapping and Single-Stepping 5-3
Conditional Breakpoints 5-3
Do Not Use Trace With System Functions 5-3
Watching Variables 5-4
Formatting 5-4
Watching the Proper Variables 5-5
Executing Functions While Debugging 5-5
Modifying Variables "On the Fly" 5-6
Removing Screen Flicker 5-6
Codeview's "Output Was Lost" Error Message 5-7
Restarting The Program 5-7
The CURRENT.STS file 5-7
HDWCLR.EXE and hdwclr() 5-8
Most Common Mistake Using C 5-8
Using the Proper Data Types 5-8
Declaring Array Sizes 5-9
PCLTS User Interface 6-1
The LTSSHEL System Shell 6-1
Privilege Levels 6-1
The Main Menu 6-1
Selecting the Test Program 6-2
Selecting a Handler Configuration File 6-3
Loading the Test Program 6-3
Verifying the Test Setup 6-3
Temporary Escape to DOS 6-4
Processing a Datalog File 6-4
Board Diagnostics 6-6
Exiting the Shell 6-6
The Development Menu 6-7
Codeview Debugger 6-7
Creating Handler Configuration Files 6-7
Long Term Calibration 6-10
System Configuration 6-10
The Test Program Pull-down Menus 6-12

THE DATALOG MENU 6-12
Datalogging to the Screen 6-13
Interrupting Datalog to Screen 6-13
Datalogging to Disk (Local or Network) 6-13
Selecting Tests to Datalog 6-14
Displaying Test Limits 6-15
Sample Datalog 6-15
THE SUMMARY MENU 6-15
Clearing the Summary Data 6-16
THE OPTIONS MENU 6-17
Start New Lot 6-17
Forcing Through Failures 6-17
Retest 6-18
Setting the Serial Number 6-18
Displaying the Test Time 6-18
Tone 6-18
User Switches 6-18
THE CONFIGURATION MENU 6-19
Handler Port 6-19
Family Board 6-19
Dlog File Path 6-20
THE ENGINEERING MENU 6-20
Bypass Cal 6-20
Collect Data 6-21
System Reset 6-21
THE FUNCTION KEYS 6-22
F1 - Test 6-22
F2 - Force Calibration 6-22
F3 - Send Summary to the Screen 6-22
F4 - Send Summary to the Printer 6-23
F5 - Datalog to the Screen 6-23
F6 - Start New Lot 6-23
F7 - Display Current Setup 6-23
F8 - Quit the Test Program 6-23
Using Ctrl-Break to Abort 6-24

Optimizing The Program 7-1

Producing a Well-Structured Test Program 7-1
Avoid using goto 7-1
Use Meaningful Variable Names 7-1
Divide and Conquer 7-1
Use Local and Global Variables 7-2
Reducing Test Times 7-2
Wait Time as a Percentage of Total Test Time 7-2
Use fast_meas() and fast_diff() 7-3
The msd() function is no longer needed 7-3
The Source Profiler 7-3

Using CONVERT 8-1

Transferring the LTS Basic File to the PC 8-1
Preparing the LTS Basic Program for Conversion 8-1
Housekeeping Code 8-2
Test Parameters Statements 8-2
Test Numbers 8-2
Regarding Subroutines 8-2
Maximum Line Width 8-3
Nested IF Statements 8-3

Line Numbers 8-3
Running CONVERT.EXE 8-3
Conversion Examples 8-5
Example 1.- Using a limit table 8-5
Example 2.- Using embedded test parameters 8-6
Example 3.- Using neither embedded test parameters nor limit table 8-7
The CONVERT.RPT File 8-7
Editing the Converted PCLTS C-Language File 8-8
Unused Variables and Statements 8-8
Deleting Test Number Increments 8-8
Variable Type Declarations 8-9
An Important Note on Array Subscripts 8-9
Replacing gotos 8-10
Other Utilities 9-1
Running Verify Setup 9-1
Correlation Criteria File Format 9-1
CORRLGEN.EXE Utility 9-2
Trial Files 9-2
Trial File Format 9-2
Reference File Format 9-3
PRINTCAL.EXE Utility 9-4
The Cal Factors 9-6
When Calibration Fails 9-7
The Error Code Word 9-7
BRDINSTL.EXE Utility 9-8
HDWCLR.EXE Utility 9-8
PCLTS System Library Functions 10-1
Functions Listing 10-1
PCLTS System Variables 11-1
Variables Listing 11-1
Troubleshooting Hardware 12-1
The DIAGNOSE.EXE Utility 12-1
Auxiliary Register Test 12-1
CRU Write Test 12-1
CRU Read Test 12-2
Timer Test 12-2
Line Synch Test 12-3
Faulty Serial Port Ribbon Cables Inside LTS 12-3
Tighten Cable Connectors 12-4
Troubleshooting System Software 13-1
Codeview Screen "Hangs-up" When Debugging 13-1
When Codeview Really Hangs-up 13-1
Not Enough Available Memory 13-1
TSRs and Device Drivers Should be Loaded High 13-2
Use an Automated Utility to Load Drivers High 13-2
Some PCNFS Drivers May Not be Loaded High 13-2
"VRD Out of Range" Error 13-2
Output Wraps-around When Using lts_printf() Function 13-3
"Fail Bin 0" and "Pass Bin 0" Problems 13-3
LTS Basic Cross-Reference Table 14-1
STDF Datalog File Format 15-1
STDF File Record Ordering 15-1
Fields Used Within Each Record 15-3
Master Information Record (MIR) 15-3

Part Information Record (PIR) 15-3
Parametric Test Result Record (PTR) 15-3
Parametric Test Description Record (PDR) 15-3
Software Bin Record (SBR) 15-4
Test Synopsis Record (TSR) 15-4
Master Results Record (MRR) 15-5
Interconnecting Cables 16-1
LTS PORT1 and PORT2 Ribbon Cables 16-1
Faulty Ribbon Cables 16-1
Measuring Cable Resistance 16-1
LTS Ribbon Cables Mapping 16-2
Emulator Extender Bracket Cable 16-4
Cables Connecting PC to LTS Tester 16-6
Gain Settings 17-1
Specifications 18-1
CPU Emulator Board (TSA-2500) 18-1
PC Requirements 18-1
Software Requirements 18-1

Overview

The PCLTS accelerator for the LTS-2020 test system

The LTS-2020 is a versatile and relatively inexpensive tester that has been around for more than 10 years. It requires minimal maintenance, has a quick test floor setup time, and delivers very reasonable performance for testing a wide variety of linear and digital parts. Overall it is a very cost-effective and capable tester within its class.

The PCLTS upgrade system brings the LTS-2020 up to date in processing speed and engineering development tools. This allows for test times comparable to today's high performance testers at a small fraction of the cost and without the need to develop new test hardware and software for existing products. It also allows the user to take advantage of today's wide range of engineering software and hardware for the PC and DOS platform.

The center of the system's software is a shell from where the user can launch the test program, view datalog files, start an engineering development session, or create handler configuration files. This allows for a centralized and controlled interaction between system modules and eliminates the need to operate from the DOS command line. All the system software is operated by means of easy-to-read pull-down menus.

The entire LTS Basic instruction set has been re-written and converted to C-language functions. The user has access to these functions from his/her test program through regular C function calls. Programs must be compiled and then linked with the PCLTS library provided. This generates an executable test program file.

ADVANTAGES OVER STANDARD LTS-2020

Production Test Floor impact

- Reduces test times to (not by) an average of 1/4 of the original test time. Potential for higher reduction (up to 1/10th) for some products. Allows for significant increase in test floor capacity with minimum effort.

- Built-in network support. Uses standard Ethernet and TCP/IP (When PCNFS software from SUN Microsystems is installed). Can be connected directly to SUN Ethernet network. System software allows for easy and transparent downloading of test programs from server to PC and uploading of datalog files to server.

- Easy to use menu-driven user interface. Fast learning curve for new users. Most operator tasks require only one or two key strokes.

- System has built-in automated setup verification system. This allows for operator validation of the test setup before testing every product lot.

- Program security. Operator does not have access to development menus, where source code could be changed. In LTS Basic it is very easy to modify a line of code at run time.

- No need to make any changes to existing test hardware (Family boards, socket assemblies, etc.). Converted test programs will functionally run in exactly the same way, only much faster.

Engineering Impact

- Software is developed in industry-standard C-language, with a powerful compiler and debugger (Microsoft C/C++). C allows for clean and structured programming and has much more powerful constructs than LTS Basic. The debugger allows for trapping, single stepping, variable and memory watching, and many other features that make program development faster and easier.

- Practically no memory limitation for program size or number of variables (as opposed to LTS BASIC).

- The high speed execution, together with an added accurate 1us-resolution timer, allow for higher timing accuracy and resolution in signal generation and measurements.

- Datalog data is saved to memory and dumped (to screen, printer, disk, or network) *after* the part is tested, as opposed to *during* testing. This avoids unnecessary warming of the part and ensures valid data. Actually, the test time remains unchanged when the datalog option is on.

- Test limits, test names, units and other parameters are optionally handled in a separate "data-sheet" editable ASCII file for simplicity, and loaded in memory by the test program at run time.

- Data can be easily transferred to other standard DOS applications on the same PC (such as spreadsheets, word processors, databases, etc.), or to MAC and SUN applications through the network.

- System generates datalog files in the industry-standard STDF format. This allows for datalog files from many different type of testers to be processed using a single application at a central server. These are binary files, as opposed to ASCII text files, and therefore occupy much less storage space.

- Test engineer has access to a large library of functions which simply do not exist on the LTS (mathematical, graphical, etc.).

- System runs on fast 486 PC platform with a large and fast hard disk. Saving and loading a file takes a fraction of a second instead of two minutes.

How the increase in speed is achieved

PCLTS User's Guide

- The original LTS-2020 has a 15-year old processor with an obsolete architecture. On the upgraded PC-LTS system, the software is run on a fast 80486 computer, which includes pipelined architecture and a math co-processor.

- The CPU emulator board can process LTS instructions by itself, without tying up the 486 CPU. This allows for parallel processing: while the emulator board is sending an instruction to the tester, the PC can be processing math calculations or executing other instructions. In the time that it takes for an average LTS instruction to be completed, the 486 CPU can execute up to ***fifty*** non-LTS instructions.

- Most of the LTS instructions have been written and optimized using assembly language. This generally allows for better speed optimization than compilers.

- The LTS bus clock (produced by the emulator) has been increased by 30%.

Conversion of Existing Basic Programs

Conversion of existing Basic programs is fairly straightforward. All the original LTS instructions have been re-written using the same name and very similar syntax.

Even though direct instruction-by-instruction program conversion will work fine, test time can be further reduced by arranging the sequence of instructions in order to take optimum advantage of the parallel processing offered by the system.

Included with the system software is a conversion program which automates the translation of LTS Basic test program files to PCLTS system C-language files. This utility will do about 95% of the conversion (leaving variable declarations and other "house-keeping" for the user to complete) enabling very quick and simple conversion of existing test program libraries.

Hardware Installation

System Requirements

The following are the specifications required for the computer where the PCLTS system is to be installed.

- IBM-compatible computer 486DX-50 Mhz (or 486DX2-66 Mhz) with ISA architecture (standard IBM-PC AT type I/O channel).
- 8 MB of RAM memory
- VGA color graphics
- Hard disk (250 MB or larger recommended for storage of test data files)
- GPIB PC-II board, from National Instruments
- Microsoft-compatible mouse
- The CPU Emulator board is a full-height (and full-length) card. Some "half-height" low-profile desktop PCs may not be able to accommodate it (in such PCs, boards are placed horizontally instead of vertically).

Installing the CPU Emulator Board

Two components need to be installed inside the PC: the CPU emulator board (TSA-2500) and the auxiliary bracket with the 25-pin connector and ribbon cable. Follow the following steps to install these two components.

1. Turn off the power to the PC and unplug the power cord from the wall outlet to prevent accidental electric shock.

2. Remove the computer cover following the procedure outlined in your computer user's manual.

3. Locate an available 16-bit (not 8-bit), full-length slot on the motherboard. The slot should also have an available empty slot to its immediate right so that the auxiliary bracket can be installed. This second slot can be either an 8-bit or 16-bit since the slot itself will not be used.

4. Remove the rear panel rail cover plates (brackets) corresponding to the two slots and save the screws.

5. Gently place the card in the 16-bit slot and align the card edge-connector gold fingers with the two connectors on the motherboard. See figure 2-1.

6. Insert the card into the motherboard sockets by slowly rocking it and applying downward pressure. This may require gently bending the board's metal bracket towards the front of the PC since sometimes the side-by-side brackets may get entangled. Make sure that the board is fully seated in the motherboard sockets and that the board's metal bracket is flush with the top of the computer's rear panel rail.

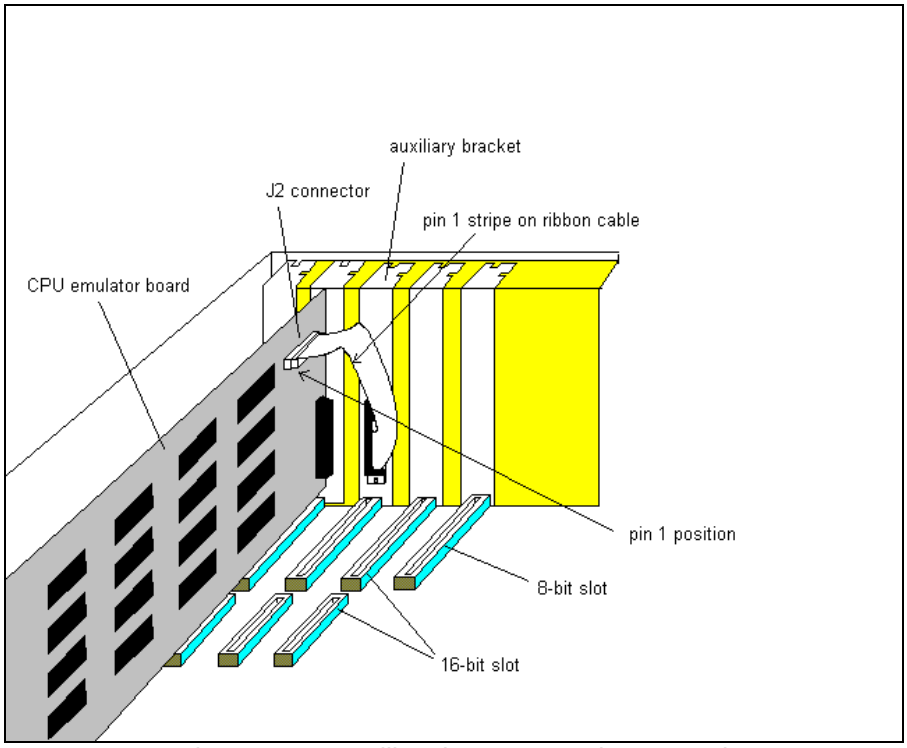


Figure 2-1 ; Installing the CPU Emulator Board

7. Secure the board to the rear panel rail by using the screw saved from step 4.

8. Take the auxiliary bracket assembly and place the metal bracket on the rear panel rail to the right of the CPU emulator board. Use the remaining screw to secure the bracket to the rail.

9. Take the IDC 26-pin connector and plug it into the header labeled J2 on the CPU emulator board. The J2 header can be found just below the white "Calyx" logo. **The connector must be plugged so that the colored stripe on the ribbon cable is towards the front of the computer**. Refer to figure 2-1 for the correct cable position. Plugging the connector is done easier if it is held at an angle so that one corner of the connector is inserted first.

10. Before powering-up the PC or putting the cover back on, inspect all connectors. Make sure the board is fully seated in the motherboard slots and that the ribbon cable connector is plugged into J2 in the right direction and all the way in.

11. Put the cover back on and power up the system and power it up.

Installing the Bus Driver Board

The LTS bus driver board (TSA-2550) replaces the original LTS CPU board inside the tester itself. Follow these steps for proper installation:

1. Power down the LTS-2020 tester and unplug the power cord from the wall outlet to prevent accidental electric shock.

2. Disconnect all cables from LTS serial ports 1 and 2. These can be found on the back side of the tester and are labeled "PORT1" and "PORT2". The terminal (connected to PORT1) will not be needed for the PCLTS system.

3. Remove the screws that hold the top of the LTS enclosure. Carefully remove the top and put it aside.

4. The system boards inside are secured in place by two metal strips that are screwed to the frame. Loosen the four screws and remove the two metal strips.

5. Identify which board is the CPU board. The CPU board has the following characteristics:

   - It is the one with the most number of ribbon cables connected to it: two cables are connected to PORT1 and PORT2 and another cable goes to the floppy disk drives.

   - It is located in the center slot of the backplane, with four slots behind it and four slots in front of it..

   - It has a daughter ("piggy-back") board with memory chips on it. This daughter board connects to the CPU board by means of a ribbon cable.

6. Use a permanent marker to mark the top of the white plastic rail where the CPU board goes. The CPU slot is unique since it does not have the same signals as the rest of the slots. This is why it must be marked. Make sure that the slot marked corresponds to the CPU board itself (not the one aligned with the daughter board). Again, the CPU slot is the middle one, with four slots in front of it and four slots behind it.

7. Also use the marker to label the ribbon cables so that they are easily identifiable. Standing at the front of the system, label them from left to right using consecutive alphabet letters starting with "A". You will need to know the order of these cables when installing the Calyx bus driver board and also if you ever want to convert the LTS system back to its original mode (two of these cables have identical connectors and it is easy to mistake them if unmarked). The labels will also help you identify which side of the connectors should face up.

8. Carefully unplug all the cables connected to the CPU board. Be gentle to avoid damaging ribbon cable connections.

9. Remove the CPU board (the daughter board will come out with it) by holding it from the card pullers and applying a lever action to them.

10. At this point it is highly recommended that a resistance and continuity check be performed on the two 26-lead ribbon cables that connect to PORT1 and PORT2 (cables labeled "A" and "B" in step 7) to the LTS CPU board. Please refer to the appendix sections for instructions on how to perform this optional check.

11. Take the Calyx bus driver board and insert it **in the same slot** the LTS CPU board was in. Make sure the board is fully seated in the backplane connector. This may require gently rocking the board while applying downward pressure. Make sure that the component side of the board faces the same direction as the rest of the boards in the system.

12. Plug the three ribbon cables labeled "A" ,"B" and "C" in step 7 into the headers labeled "J2", "J1" and "J3", respectively, on the bus driver board.

Table 1-1: Bus Driver Board Connections

Ribbon Cable Label	Board Header Label	Number of Pins
A	J2	26
B	J1	26
C	J3	20

Running the Diagnostics Program

Do not install the LTS-2020 console's cover back on yet. Proceed to install the system software and then come back to this section.

Once the PCLTS system software has been installed, type LTSSHEL /P1 to load the system shell in the engineer privilege level. Select the ***Run Board Diagnostics*** option from the main menu (option 8) and then select the ***Perform All Tests*** option (option 6). The system will perform a sequence of five tests which check the integrity of the CPU emulator board, bus driver board, and interconnecting cables. These tests do not check the LTS-2020 console itself.

Please refer to the **Troubleshooting Hardware** chapter of this manual for details on the diagnostics utility:

Setting Up the Printer

Setting the Printer's Character Set

Summary sheets, datalog printouts, and limit tables contain extended ASCII characters which are generally not part of the default printer character sets. In order for them to be printed properly, the printer's character set must be set to the appropriate one.

Consult the printer's manual to determine the character set to choose. Generally the manual will show listings of the actual characters in each set. Pick the one that includes the characters "»", "¼", "É", "È", "º" and "Í". In most printers, the character

set may be easily changed and saved using the buttons in the front panel. Some older models require changing dip-switches settings.

A good test to check for the proper printer character set is to print a summary sheet (hit F4 from the test program screen). The frame in the header of the printout should look like this:

```
SUMMARY SHEET
```

```
        Product: DUMMY DUT
    Operator ID:                      Test Mode Code:
 Station number: 0                 Test Conditions Code:
         Lot ID:                           Node Name:
       Job Name:
Lot Description:
           ----- Report Generated on 06/03/94 at 20:25:09 -----
```

It is a good idea to pick a character set with smaller characters so that lengthy datalog printouts require less pages and so that wide limit tables may fit across the page. For limit tables with many sets of limit types, it may be preferable to pick a landscape font (printed horizontally).

The Printer Port

Parallel Printers

The PCLTS system expects the printer to be found at LPT1 (parallel port #1) of the PC. Some PCs have two parallel printer ports, denominated LPT1 and LPT2 for ports 1 and 2, respectively.

Serial Printers

If a serial printer is being used, then it will be connected to one of the PC's serial ports: COM1, COM2 or COM3. In order for the PCLTS system to access the printer, the serial port to which it is connected must be re-directed to LPT1 by typing the DOS command:

```
MODE LPT1=COM1
```

If the serial printer is connected to port 2, replace COM1 for COM2 in the above command. This command may be included in the AUTOEXEC.BAT file for more convenience.

Software Installation

Software Requirements

- MS DOS version 6.0 or later
- Microsoft Visual C++ Professional Edition V1.0, V1.5, or V2.0
- Microsoft Windows is needed to install Visual C++ (but not needed to run)

Microsoft's Windows should be the first program to be installed since Visual C++ requires it for its installation..

Then you may install the C compiler and the PCLTS system software, in that order. Please follow the procedure described below.

Installing Microsoft Visual C++ Compiler

Microsoft's package is a complete development kit for creating Windows applications using C++. The PCLTS system uses only a small fraction of the entire system; namely, the C compiler itself, the linker, the DOS version of the Codeview debugger, the DOS profiler and the large model library. Unfortunately, Microsoft does not offer all these modules in a separate package.

Note that Microsoft Windows is required to install version 8 of the C compiler (even if Windows will not be used to run it).

Microsoft Visual C++ V1.0 and V1.5

To begin installation of the C compiler, run Windows and use the *File* menu to *Run* the SETUP program from the floppy drive. Shortly after loading of SETUP, the *Installation Options* screen will appear, prompting the user to select the file groups to install. It is very important to select the correct installation options, or else the PCLTS system will not function properly. Please be careful to select the following options:

To de-select a group, click on the "X". All the groups will be selected by default, but only the following groups are needed:

- **Microsoft C/C++ Compiler**
- **Run-time Libraries**
- **Online Help Files**
- **Tools**

Installing and using the *Visual Workbench* is optional and is left up to the user. Calyx recommends compiling and linking from the DOS command line using the batch files provided (**compile.bat** and **linkit.bat**). *Visual Workbench* is an excellent environment for developing Windows programs, but its flexibility and large number of

options may make development of simple DOS test programs unnecessarily complicated.

Then, on that same screen, click on the *Libraries* button to go to the *Library Options* screen. There will be three blocks of options: *Memory Models*, *Targets* and *Math Support*. The following should be the only options selected in these blocks:

In the *Memory Models* block:

- **Large/Huge**

In the *Targets* block:

- **MS-DOS .EXE Files**

And in the *Math Support* block:

- **Emulation**
- **80x87**

Click on the OK button to go back to the *Installation Options* screen. Then click on *Tools* to go to the *Tool Options* screen. There will be two blocks of options: *Windows Tools* and *MS-DOS Tools*. The following should be the only options selected in these blocks:

In the *Windows Tools* block:

- No options from this block should be selected.

In the *MS-DOS Tools* block:

- **Codeview**
- **MS-DOS Profiler**

Click on the OK button to go back to the *Installation Options* screen. Then click on *Continue* to continue the installation.

Microsoft Visual C++ V2.0

This version is only available on CD-ROM. It includes in four subset groups of software packages: *Visual C++ 2.0, OLE Custom Control Development Kit, Visual C++ V1.51, and WIN32 Operating System*. Of these four, the PCLTS system uses only ***Visual C++ V1.51***. (Confused by now?).

To begin installation, run Windows and use the *File* menu to *Run* the SETUP program from the CD-ROM drive (which should be drive letter D: or E:). The Visual C++ Master Setup window will appear, prompting the user to select from the four subset groups available. **Click on the third button from the top: Visual C++ V1.51.**

When the *Installation Options* screen appears, **click on the Custom Installation button (middle button)**.

When the Custom Installation window appears, select the file groups to install. It is very important to select the correct installation groups, or else the PCLTS system will not function properly. Please be careful to select the following groups:

To de-select a group, click on the "X". All the groups will be selected by default, but only the following groups are needed:.

- **Microsoft C/C++ Compiler**
- **Run-time Libraries**
- **Tools**
- **Online Help Files**

Installing and using the *Visual Workbench* is optional and is left up to the user. Calyx recommends compiling and linking from the DOS command line using the batch files provided (**compile.bat** and **linkit.bat**). *Visual Workbench* is an excellent environment for developing Windows programs, but its flexibility and large number of options may make development of simple DOS test programs unnecessarily complicated.

Click on **MFC OLE** and **MFC Database** *New Features* file groups to de-select them, since they are not needed. Then click on the **Continue** button.

Finally, answer YES to the prompt *Do you want to Install Phar Lap's DOS Extender Lite?*

This completes the C compiler installation.

Please refer to Microsoft's manuals if you have any doubts when installing the compiler.

Microsoft Visual C++ V1.52 and Visual C++ V4.00

For these two versions follow the same installation instructions as described above (proceed to install by clicking the **Custom Installation** button). Microsoft compilers come in two forms: *Standard Edition* and *Professional Edition* packages. The Standard Edition is sufficient for developing PCLTS test programs (the Standard Edition is a subset of the Professional Edition and sells for about one fourth of the price).

A Couple of Important Details

Important Note There are several environment variables that need to be set in the system before invoking the compiler from the DOS command line. Microsoft provides a batch file called MSVCVARS.BAT to set these variables before compiling a program. This batch file should be put as part of the AUTOEXEC.BAT file for more convenience. The MSVCVARS.BAT file can be found in the MSVC\BIN directory. Please refer to the **MS-DOS Configuration** section of the Microsoft user's guide for more details.

Version 8 of the Microsoft C compiler (called CL) is included as part of *Microsoft Visual C++ Professional Edition* software. The compiler version should not be confused with the entire development system's version. The compiler version is displayed on the DOS screen every time that the compiler is run from the DOS command line.

Installing the PCLTS System Software

If the PC will be used for program development (as opposed to test floor program execution only), then the C compiler should be installed prior to the PCLTS system software. The C compiler is not needed for running the test programs, only for development and debugging.

To install the PCLTS system software, insert the installation disk number 1 in drive A or B and type *install*. The INSTALL program will prompt the user for the target drive where the PCLTS system files will be installed. Normally this would be the C: drive.

The program will then prompt for a virtual drive on the network server disk. This drive will be used for storing datalog files when the user selects to datalog to the network server.

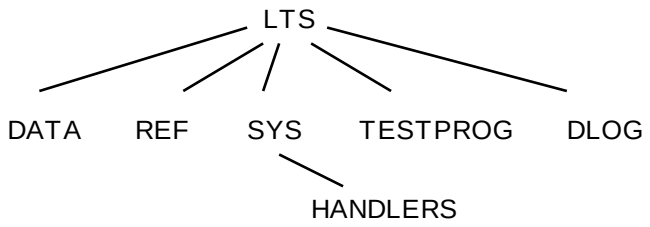
Install then asks the user if the C compiler has already been installed on the PC. The compiler should be installed first, or else INSTALL will not copy the PCLTS system library and *include* file. If the system will not be used for development (an example of this would be a system destined to a production test floor), then installing the compiler is not necessary.

If the user answers "Y" to the question above, the system then prompts for the C compiler files path, normally C:\MSVC. This is used by INSTALL to decide where it will store the system library file (PCLTS.LIB) and include file (2020LANG.H). It will store these two files in the LIB and INCLUDE sub-directories of the path entered by the user.

Defaults will appear for all the prompts, but these may be edited on the screen by the user as needed.

The PCLTS Directories Tree

The install program will create a directory tree on the target drive under the LTS parent directory. The following is a graphical representation of the tree:



PCLTS Directories Tree

The purpose of each on of the sub-directories is the following:

DATA The PCLTS system has the capability of producing data point files that can be processed by a graphical/mathematical analysis program to do FFT or graphic plots such as linearity plots for converters. This option is enabled by using the ***Engineering*** and ***Collect Data*** options of the PCLTS pull down menus, and works

in conjunction with the *record_data_point()* function. Data collection files have the extension DAT (*****.DAT).

This directory is NOT for datalog files.

REF Device reference files (*****.D*****) and correlation criteria files (*****.COR) should be stored in this sub-directory. These files are used when the ***Verify Setup*** option is invoked from the LTSSHEL menu.

SYS Contains executable system modules called by the PCLTS system.

SYS\HANDLERS This directory stores the handler configuration files (*****.HND). These files contain timing, voltage level, and sort line parameters for all the different handlers.

TESTPROG All test programs (*****.EXE) and limit files (*****.LIM) should be stored in this location. The LTSSHEL program will look in this directory for them.

DLOG This is where the datalog files (***.**STD) will be stored when the user selects the ***Datalog to Disk*** option from the *Datalog* menu.

Note that the above directories are only a system default and may be changed by using the ***System Configuration*** option (option 6) of the ***Development Menu*** option (option 7) of the LTSSHEL menu. They can be replaced by any directory in any drive, including a virtual disk on a network server drive. Also, the same directory may be used for two or more purposes.

The LTSSHEL.INI Configuration File

The LTSSHEL.INI file is an ASCII file that contains settings used by the LTSSHEL program and the application test programs every time they are executed. This file must be in the C:\LTS\SYS directory at all times. In fact, even if the user installed the PCLTS system files onto the D: drive, the install program will still create the C:\LTS\SYS directory and store the LTSSHEL.INI file there (with all the other system files on drive D:). The first time that the LTSSHEL or a test program is loaded, this file is created by the system using default settings.

The LTSSHEL.INI file can be modified by the user at any time by using the ***Test Development*** option (option 7) of the LTSSHEL menu and then selecting the ***System Configuration*** option (option 6). Only 10 of the 11 lines of this file are modifiable (line 3 is not).

The configuration file consists of the following 11 lines:

Line 1. The name of the last test program that was selected from the LTSSHEL program. Note that this name does not include the ".EXE" extension.

Line 2. The name of the last handler configuration file that was selected from the LTSSHEL program, without the ".HND" extension.

Line 3. The time corresponding to the last successful hourly system calibration. This time is stored as a "long int" type, and is the number of seconds after 0:00:00 Jan. 1, 1970 (following DOS time convention).

Line 4. Path name for storing datalog files on the local PC hard disk.

Line 5. Path name for storing datalog files on the network server hard disk.

Line 6. *This line is no longer used by the system* (it was used in earlier versions and was kept in the file for back-compatibility).

Line 7. Path name for storing PCLTS system files.

Line 8. Path name for storing executable (".EXE" extension) test programs. The LTSSHEL program will only load test programs that are found on this path.

Line 9. Path name for storing handler configuration files (".HND" extension). The LTSSHEL program will only load configuration files that are found on this path.

Line 10. Path name for storing graphical analysis data to be processed externally by another software tool (like DaDisp or Excel). This is not a path for datalog data.

Line 11. Path name for storing device reference files when using the ***Verify Setup*** option of the LTSSHEL program.

Important: Note that all the file paths in this file end with a backslash character ("\"). This is seen only if the file is edited directly with a text editor (not recommended). It is not seen if the ***System Configuration*** option of the ***Development Menu*** is used.

The INSTALL program will create the configuration file using the drive names entered by the user. It will be similar to this:

```
<<NO_SELECTION>>
<<NO_SELECTION>>
0
C:\LTS\DLOG\
F:\LTS\DLOG\
C:\LTS\SOURCE\
C:\LTS\SYS\
-- unused line (BUT DON'T DELETE) --
C:\LTS\SYS\HANDLERS\
C:\LTS\DATA\
C:\LTS\REF\
```

The first three lines will be properly updated by the system immediately after the first execution of the LTSSHEL program.

Even though the LTSSHEL.INI file is an ASCII file and may therefore be modified using any standard text editor, it is safer to use the ***System Configuration*** option under the ***Development Menu*** option on the LTSSHEL program. All the path names in this file may be modified to settings other than the default ones. Any path name modifications will remain unchanged until the user edits the file once again. The system will update only the first three lines as needed.

Maximizing Available DOS Memory

PCLTS executable test programs are relatively large since they carry the user interface embedded in them. This implies that they occupy a large amount of memory at run-time. Also, a large amount of RAM is allocated dynamically (as the program is being executed) for storing datalog and summary information. For this reason, test programs need as much free DOS memory as possible at run time. Unfortunately, the typical PC setup has a number of memory-resident programs and utilities (device drivers and TSRs) occupying the lower 640k of DOS memory. The available DOS memory in a PC can be seen by typing ***MEM*** from the DOS command line (MSDOS Version 4.01 and higher).

Memmaker, a memory manager utility included with DOS 6.0 and higher, will automatically load device drivers and TSRs in high memory. For details on running this consult your DOS manual.

There are also two other excellent third-party memory managers on the market: 386Max and QEMM.

386MAX Memory Manager

386MAX, a memory manager from Qualitas, is highly recommended as a tool for maximizing the available DOS memory. If you are using this, after installing the C compiler, run the **maximize** program from the 386MAX directory. This is an automated system configuration that will examine your PC's setup and load as many memory resident programs as possible in "high" memory (RAM memory between 640K and 1 Megabyte). This will free up the maximum amount of DOS memory (lower 640K) that is possible. It is advisable to have installed all your memory resident programs (device drivers and TSRs such as the mouse driver, disk cache and others) prior to running **maximize**, so that they can all be moved to high memory. The PCLTS system requires over 600K of available DOS memory to insure that very large test programs will run properly. Please refer to the 386MAX manual provided with the C compiler for more details.

Notes: i) Some (very few) PCs experience problems when **maximize** is run with the option to recover ROM BIOS memory set to YES. Please run the program again with this option set to NO if you experience any problems. Once again, please refer to the 386MAX documentation for more details.

ii) If Sun Microsystem's PCNFS is being used for networking, be aware that some of the smaller PCNFS device drivers may not be loaded "high". Please refer to SUN's documentation for more details.

Quarterdeck's QEMM

Another option for a good memory manager is QEMM, by QuarterDeck. QEMM has a utility equivalent to 386MAX's **maximize,** called **optimize**.

Creating a Test Program

The Test Program

Creation of a Test Program

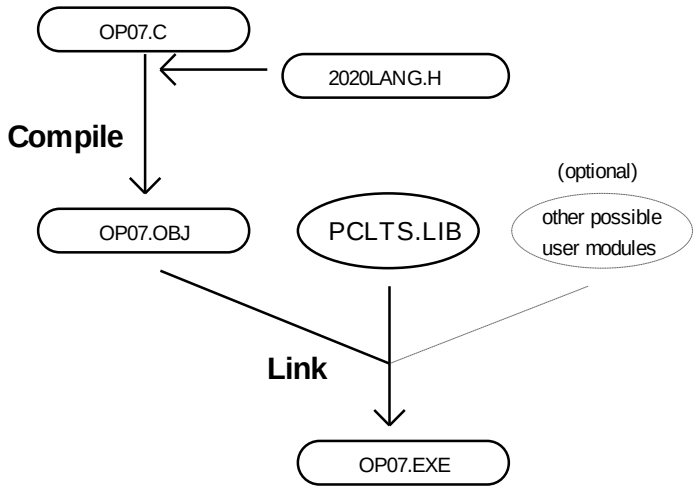
The test program is an executable file (".**EXE**" extension) which is the result of compiling the test code written by the user and linking it with the PCLTS.LIB library and possibly other user libraries. C-language programs always contain a function called *main()*, which acts as the central body of the program, and from which other functions are called. In the PCLTS system, the *main()* function is part of the PCLTS.LIB library, and the test program code that the user writes is merely a function called *test_program().* This *test_program()* function is called by the system's *main()* function at the time the "**TEST**" button is hit.

Of course, other user functions may be called from within the *test_program()* function.

The include file 2020LANG.H contains important system function prototypes (needed for all C-language functions), global variable declarations, and constant definitions. This file should be "included" in the test program source file by using the pre-processor directive

```
#include <2020lang.h>
```

at the beginning of the source file. The system installation program copies this include file to the INCLUDE sub directory of the C compiler directory.



Creation of a Test Program

Recording the Test's Parameters and Results

Once the final result for a particular test within the program is calculated, this value, together with the test name, limits, units, fail bin and formatting decimals, must all be recorded by the binning and data logging system. In the standard ADOS LTS Basic, this was done setting some variables and the calling the 32000 subroutine:

```
2000 VL=RES/1000
2100 $TAG[0]="Input Resistance":: FCLASS 1 TO 8
2200 $FMT[0]=" SS.999":: LL=2.45:: UL=2.65:: $UNT="Kohms"
2300 GOSUB 32000
```

In the PCLTS system this is done by calling two functions and passing the test parameters as function arguments:

```
value=res/1000;
test_parameters(value,"Input Resistance","Kohms",2.45,2.65,"3",9);
bin_current_test(value);
if(check_fail()) return;
```

The first parameter in the *test_parameters()* function call is the result that is being logged. This may be any variable name or an actual number (unlikely). It is followed by the test name and units strings. These may also be string variables instead of the actual literal strings. These are followed by the lower and upper limits, which, again, may be literal values, variables, or even complex expressions. Then comes the number of formatting decimals, in string form. This is the number of significant figures to be displayed left of the decimal point on the result. It is only for displaying purposes. The actual result value is stored with maximum precision. The last parameter is the fail bin. It is the actual fail bin (9, in the example), rather than what is left after disqualifying the fail classes (1 to 8, in the example).

The *test_parameters()* function must be followed by *bin_current_test()*. The name is self-explanatory. It is important to pass the same result parameter as in the *test_parameters()* function. In the example above this parameter is "value".

Finally, the *check_fail()* function must be called to provide an exit point when the test program is being run in the "stop on first failure" mode. This function does not have arguments.

If a limit table is being used, then the equivalent code is even simpler:

```
value=res/1000;
auto_test_test_parameters_and_bin(value);
if(check_fail()) return;
```

For this case, all the test parameters would be found in the limit table file and would not be part of the test program.

tnum is Incremented Automatically

After every call to the *bin_current_test()* function (or, if a limit table is being used, after every call to the *auto_test_parameters_and_bin()* function) the system variable *tnum* is automatically incremented. Therefore, there is no need for the user to do this in the test program (unlike the original ADOS LTS Basic programs).

Of course, *tnum* may be incremented by more than 1 or set to any value at any point within the test program when the situation calls for it.

Binning

Differences with LTS Basic

The device classification system in the PCLTS system works basically in the same way as the ADOS LTS Basic system. There are three differences however:

- The total number of possible bins in the PCLTS system is 32 instead of 48.
- When a part downgrades, the *fail* flag is not set, even though the test failed one of the pass bins when it downgraded. Only the *test_downgraded* flag is set. If a limit table is being used, the system will automatically set this flag as needed. When a limit table is not used, it is the user's responsibility to set it when required (see section below on *test_downgraded* variable).
- The fail bin reported for a failing part always corresponds to the first test failed, regardless of subsequent tests failing when the program "forces through" failures. In this way the fail bin is the same when stopping on the first failed test and when "forcing through".

Some minor differences also apply to the syntax for calling the *pclass()*, *aclass()* and *fclass()* functions. See the **PCLTS System Library Functions** for more details.

Setting the Fail Bin

In the PCLTS system there is no need for calling the *fclass()* function since the fail bin for a given test is set by passing it as an argument in the *test_parameters()* function. It is passed as the seventh (last) argument, and it must be the actual fail bin number. In LTS Basic, setting the fail bin for a given test to, for example, 15, is done by calling *FCLASS 1 to 14*. In the PCLTS system, this is done by actually passing the number 15 as an argument:

*test_parameters(?,?,?,?,?,?,**15**)*;

When a limit file is being used, the fail bin for each test is included in the limit table and is not part of the test program source code.

Disqualifying Passing Bins When Down-grading

When the *bin_current_test()* function is called after each test in the program, the system will automatically disqualify the necessary bins based on the pass/fail limits set by the *test_parameters()* function.

It is possible for the user to set more than one passing bin when a product has several different performance grades. If a limit table is not being used, then it is up to the user to disqualify higher passing bins when the part results in a passing device that does not meet bin 1 specifications (a downgrade).

Just like in the LTS Basic system, in the PCLTS system bins can be disqualified by toggling bits on the *pclasv* system variable. Instead of using the BIT statement, C-language employs bit-wise operators such as logical ANDs and ORs and bit shifting. However, a much easier and safer way to modify the *pclasv* variable is to call the *disqualify_bin()* function:

```
disqualify_bin(3);    /* disqualify bin 3 */
```

The *test_downgraded* variable

In addition to disqualifying higher bins, the *test_downgraded* variable must also be set to a TRUE logic value when downgrading a device. This will ensure that the device is included in the summary sheet as a downgrade part, and that the datalog shows a "**D**" by the test that caused the downgrading.

Again, if a limit table is used, downgrading is automatically performed by the system and is totally transparent to the programmer. There is no need to include any downgrading code in the test program.

Down-grading Example

The following is an example of "manually" downgrading a part when no limit table is being used:

```
/* These parts have three grades:

     Grade A with Vos within 1mV (Bin 1, top grade)
     Grade B with Vos within 5mV (Bin 2)
     Grade C with Vos within 10mV (Bin 3)
                                                */

pclass(1,3);   /* three passing bins in the program (three grades)*/
bin3_lo=-10; bin3_hi=10;     /* pass/fail limits (bin 3) */
bin2_lo=-5; bin2_hi=5;       /* bin 2 limits */
bin1_lo=-1; bin1_hi=1;       /* bin 1 limits (top grade) */
meas(6,NOSYNC);
value=res*18.34;
test_parameters(value,"Offset Voltage", "mV",bin3_lo,bin3_hi,"2",28);
bin_current_test(value);  /* pass/fail limits were passed above */

/* ---- Now let's check for downgrades: --- */

if((value>bin3_lo)&&(value<bin3_hi_lim)){ /* didn't fail. Downgrade? */
  if((value<bin1_lo)&&(value>bin1_hi)) disqualify_bin(1); /* not a 1 */
  if((value<bin2_lo)&&(value>bin2_hi)) disqualify_bin(2); /* not a 2 */
}
```

Function Bodies Required in the Test Program

There are four functions that must always be included in the test program source file. These are:

- *test_program()*
- *define_bin_descriptions()*
- *user_before_testing_part()*
- *user_after_testing_part()*

Of course, the test program source file may contain as many other user functions as he or she may need.

test_program() function

All C-language programs contain a function called *main()*, which acts as the central body of the program, and from which other functions are called. In the PCLTS system, the *main()* function is part of the PCLTS.LIB library. When *main()* runs, the main test screen with the pull-down menus is presented and the system waits for the user to make a selection or hit RUN. When F1 (RUN) is hit, *main()* calls the *test_program()* function and the test program is executed. Once all the tests are executed, *test_program()* is exited and control returns to *main()* again.

The *test_program()* function may call other user-defined "lower level" functions which may contain code for tests, with the only condition that the line of code

```
if(check_fail()) return;
```

that must go after every test is always put in the *test_program()* function, even if the tests themselves are in lower level functions. This function does not return a value and it is therefore declared with type *void.*

The *test_program()* function has the following form (assuming a limit table is being used):

```
/*******************************************************/
void test_program()
{
/* local variable declarations go here */
double value;

  /* ---- Test #1 ---- */

  value=???;
  auto_test_parameters_and_bin(value);
  if(check_fail()) return;

   /* ---- Test #2 ---- */

  value=???;
  auto_test_parameters_and_bin(value);
  if(check_fail()) return;

  /* ...more tests...*/

return;  /* return from test program */
}
```

define_bin_descriptions() function

This function is called by the system before executing the test program for the first time, setting the *bin_descriptions()* array of strings to the names provided by the user. Both passing and failing bins should be described. The descriptions are used in the summary sheet.

Example:

```
/*************************************************/
void define_bin_descriptions()
{
   /* only used bins need to be defined */

   bin_descriptions[1]="Grade A Part";
   bin_descriptions[2]="Grade B Part";
   bin_descriptions[8]="Supply Current";
   bin_descriptions[15]="Offset Voltage";
   bin_descriptions[18]="Output Leakage";
   bin_descriptions[23]="Functionality";
   bin_descriptions[24]="Input Resistance";
   bin_descriptions[27]="Power Supply Rejection Ratio";
   bin_descriptions[29]="Slew Rate";
   bin_descriptions[30]="Output Current";

   return;
}
```

This function must be used even when a using limit table.

user_before_testing_part() and user_after_testing_part() functions

When the RUN key is hit, the system calls the *user_before_testing_part()* function immediately before calling the *test_program()* function. And immediately after *test_program()* is done, u*ser_after_testing_part()* is executed. The bodies of these functions must be included in the test program source file or else the linker will generate an "unresolved external" error indicating that it did not find them.

These functions should be used by the user to execute any "housekeeping" code that is needed before and after running the test program. Even if the user has no instructions to be executed in these functions, they must still be included in the test program source file (with a simple *return* in them). The section **An Example of a Simple Test Program** includes a sample test program with these two functions at the end.

Other Required Include Files and Definitions

Include Files

Depending on what functions from the Microsoft libraries will be called from the source file, there are "include" files that must be listed at the beginning of the file. The best way to determine which include files are needed is to consult Microsoft run-time library manuals. Together with the description of each function is a listing of the include files that should appear in the program if that particular function is to be used.

The following include files should be listed as a minimum:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
```

```
#include <conio.h>
```

In addition to include files for the Microsoft functions, the file ***2020lang.h*** must be listed to cover the function and variable definitions of the PCLTS system. The "define" directive for the EXTERN constant also shown below is required as well.

```
#define EXTERN extern
#include <2020lang.h>
```

The following "define" directives are recommended but not required. The first two allow using the constants TRUE and FALSE in the test program as logic true and false. Finally, if the string copy shown in the user_before_testing_part() function in the sample test program below is used, then the PRODUCT_ID constant should be initialized to a string with the type of part being tested.

```
#define TRUE 1
#define FALSE 0
#define PRODUCT_ID "DAC-1234"
```

Variable Declarations and Function Prototypes

The top part of the source file should also contain prototypes for all the functions that the user may be adding to the program. Even though prototypes are not required for some functions, it is a good idea to always have one for every function or bizarre run-time errors may occur.

Important Note: The four required functions listed above have already been prototyped in system files. The user should only incorporate prototypes for user functions that he or she may be adding to the test program.

Finally, any global variables in the program must also be declared at the top of the source file, outside of the *test_program()* function. It is a good idea to minimize the use of global variables to avoid potential mistakes. Passing parameters as function arguments is a better and more structured way of sharing data among functions. Variables should be declared as local whenever possible. Remember that local variables must be declared inside function bodies and that their scope is limited to that function only.

Using a Limit Table

When a limit table is used, all the data sheet parameters such as test name, units, limits and lower grade information are kept in a separate file, reducing the test program source file only to code directly related to setting up the tests and taking measurements.

To use a limit table, a file must be created with all the test parameters for every test in the test program (see **Limit File Format** section below). Then, instead of calling the *test_parameters()* and *bin_current_test()* functions after every test, only the *auto_test_parameters_and_bin()* function must be called. These are the only differences between a program that uses a limits table and a program with embedded limits.

Some of the advantages of using a limit table are:

- The limit file is a separate ASCII file that may be quickly modified when changing limits, without the need for re-compiling the test program.
- The limit file may be printed and used as an easily readable document that shows all the test parameters at a glance, and it is a guarantee that those are the actual parameters being used every time the test program is run.
- The test program source code is easy to read since it only contains code directly related to setting up the actual tests.
- It makes binning very easy.

When a limit table is used, there is no need to disqualify passing bins in the test program (when a downgrade occurs). It is all done automatically by the system software.

Loading the Limit Table

If a limit table will be used, the user must indicate so by entering a limit type name in the **Type of Limits to Load from Limit Table** field of the *Operator Data Entry* dialog window. This dialog window appears on the screen the first time a test program is run. If nothing is entered in this field, the system assumes that no limit table will be used and will then expect the test program to use the *test_parameters()* and *bin_current_test()* functions to specify the test data and bin the tests.

The limit table is read from the disk and loaded into memory immediately before executing the test program for the first time. The data is kept in memory until the program is terminated.

File Naming Convention

Limit table files must have the same filename prefix as the test program, but with the "**.LIM**" extension. For example, if a limit table is being used with the test program OP_07.EXE, then it must be named OP_07.LIM.

This file must reside in the directory designated by the user for the executable test programs.

Limit File Format

Since the limit file is an ASCII text file, it can be edited using any text editor, like DOS' EDIT.EXE program. However, it is highly recommended that an editor with vertical column cut and paste capabilities be used (such as Microsoft's PWB provided with the older version 7.0 of the C/C++ compiler). To enable its "box" mode use the Edit menu. The limit table frame consists of extended ASCII "graphic" characters which can be typed in by pressing the "**Alt**" key at the same time as the ASCII code of the character. The character codes used are 169, 170, 179, 180, 181, 191, 192, 193, 196, 198, 205, 216, 217 and 218. If the table does not have this frame, errors will result when the system reads it into memory, since key characters of the table are used to detect boundaries. The best way to create a limit table is to start with the file SAMPLE.LIM and edit it.

Limit Table

Product : XXX-1234
Rev : 01-01-94
Engineer: John Doe

Pass Bin	Test# Beg	End	Test Name	Units	Fmt Decs	Fail Bin	HOT_TEMP MIN	MAX	QA MIN	MAX
1	1	1	Output Voltage	V	2	25	4.5	5.5	4.9	5.1
1	2	2	Input Resistance	Kohms	3	19	1.8	4.2	2	4
2							1	5	1.5	4.5
1	3	3	Supply Current	mA	1	6	15	23	18	20
1	4	11	Digital Input Current	nA	0	30	0	300	0	50
1	12	12	Power Supply Rejection	PPM/V	0	23	0	50	0	40

Table 1
A Limit Table Example

Table Rows

In general, each row of the limit table represents a test in the test program. There are two exceptions:

The first exception is when a single row is used to cover a range of test numbers. This can be done when all tests in that range have the same test parameters (same name, units, limits, etc.). An example of this could be digital input current tests in the eight data pins of a DAC. See tests 4 through 11 in the example above (Table 1).

The second is when there are lower grade bins for a given test. The limits for the lower grade pass bins are entered in consecutive rows from lower bin (higher grade) to higher bin (lower grade). In the example above, test number 2, input resistance, has a second row for its bin 2 (downgrade) limits. When downgrade rows are used, only the pass bin and limit columns need to be filled for those rows, since the rest of the information (test name, units, etc.) would be the same as the bin 1 entries for that test.

The following rules must be followed when using downgrade rows in a limit table:

- Downgrade rows must always be entered from lower to higher pass bin
- They must follow a regular pass bin 1 row for the same test
- Downgrade rows must have consecutive pass bin numbers. They do not need to cover all the pass bin numbers, though.

As an example, assume a test program has 4 passing bins: 1 through 4. The following two limit table downgrade entry examples would be valid:

1	9	9	Input Resistance	Kohms	3	19	1.8	4.2
2							1	5

1	17	17	Offset Voltage	mV	1	28	4.5	5.5
2							4	6
3							3	7

Valid Downgrade Rows

The following would be **invalid** (non-consecutive pass bins, since it skips from 2 to 4 without going through 3):

1	9	9	Input Resistance	Kohms	3	19	1.8	4.2
2							1.5	4.5
4							1	5

INVALID Downgrade Rows

The case shown in the example of the invalid rows above may arise when no bin 3 limits are desired for that particular test. This may be legally done by adding a bin 3 row with the same limits as the bin 2 row.

Table Columns

Pass Bin

The *Pass Bin* column indicates the pass bin number for each row of limits. If the test program has only one pass bin, then all entries in this column will be "1". For programs with more than one pass bin, this column contain a lower grade bin number.

Beginning and Ending Test Numbers

Normally, these two numbers will be the same and will correspond to the test number for that row of test parameters. Two numbers are used to allow for range of tests sharing the same test name, units, limits etc. If this is the case, the first number will be the starting test number in the range and the second will be the ending test number.

Test Name

The name for the test in that row. It may be at most 35 characters in length.

Units

The units name for the test in that row. It may be at most 6 characters in length.

Analogous to the **$UNT="mV"** type of statements in LTS Basic.

Formatting Decimals

This digit represents the number of significant figures to be displayed to the right of the decimal point on the datalog. If 0 is used, no decimals will be displayed. The formatting decimals only affect the displaying of the results since the actual data in the datalog file is stored with maximum precision.

Valid numbers are 0 through 6. The entire field size for displaying the result is 8, so when using floating point numbers, the user must consider the largest result expected and allow an amount of decimals such that the digits in the integer portion plus the decimal point plus the number of decimals does not exceed 8.

Analogous to the **$FMT="SS.999"** type of statements in LTS Basic.

Fail Bin

Indicates the fail bin for the test on that row. Must be a number from 1 to 32, since there are only 32 bins in the PCLTS system.

Analogous to the **FCLASS=1 TO 12** type of statements in LTS Basic.

Limit Pair Columns

There must be at least one column of min/max limit pairs, with the maximum being forty. A given column contains the minimum and maximum limits for the test in each row, with the minimum limits first.

More than one limit pair column may be included when a different set of limits is needed for the same test program. An example of this a product that is testes at room temperature and hot temperature. Each limit pair column must have a label (name) which can be any string, as long as it is a single string. If more than one word is used, then the words must be linked with underscores to form a single string (for example HOT_TEMP).

When the test program is run, the user must enter the limit label desired at the **Type of Limits to Load from Limit Table** prompt in the Operator Data Entry dialog window. This entry must match the label on one of the limit pair columns in the table.

Important note: If the ***Verify Setup*** tool of the LTSSHEL is used, then the limit table must have a limit pair column named QA.

An Example of a Simple Test Program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include <conio.h>

#define EXTERN extern
#include <2020lang.h>
#define TRUE 1
#define FALSE 0
#define PRODUCT_ID   "DAC-1234"    /* the device name */

/* ------ Function Prototypes ------- */
void auto_range_meas();
```

```
void auto_range_diff();
int  calibrate_DAC_family_board(void);

/* ------ Global variables ------- */

double  a6_offset;
double  a6_gain
int     fam_brd_cal_passed;


/*************************************************************************/
void test_program()
{
long int i;
double value;

  if((initial_run)||(sys_calibration_ocurred)||(!fam_brd_cal_passed)){
                     /* cal F.B. 1st time & every time sys cal occurs */
     if(!calibrate_DAC_family_board()){
       fam_brd_cal_passed=FALSE;
       return;              /* cal failed, exit test program */
     }
     else{
       fam_brd_cal_passed=TRUE;  /* cal OK */
     }
  }

  pclass(1,2);     /* two pass classes (bins) */
  gon(SA); vsa(15); gon(SB); vsb(5);  /* power up device */
  select(LA,GD); noise(0.001);

  /* ---- Test #1 ---- */
  vrd(10);
  dcb(0xa2); ddb(0xf3a1);  /* connect relays */
  lts_wait(2);             /* wait 2 ms */
  meas(3,NOSYNC);
  if(alarm) auto_range_meas();
  value=(res*a6_gain-a6_offset+rdvo);
  auto_test_parameters_and_bin(value);
  if(check_fail()) return;

   /* ---- Test #2 ---- */
  vrd(0.5); ddb(ddb_value & 0xafc2);
  vsa(5); dcb(0x13);               /* change relays */
  lts_wait(2);             /* wait 2 ms */
  null(2,SYNC);
  lts_wait(1);
  dcb(0x14);
  diff(2,SYNC);
  if (alarm) auto_range_diff();
  value=res*1000;   /* result in mV */
  auto_test_parameters_and_bin(value);
  if(check_fail()) return;

  /* ...more tests would follow...*/


return;  /* return from test program */
}
/*************************************************************************/
int calibrate_DAC_family_board()
{
char *line1="The DAC family board has failed calibration!!";
char *line2="...test program will not be run.";
char *line3="";
int ideal_A6_gain,success=TRUE;
double x,g;

  ideal_A6_gain=6;
```

```
  /* set a6_gain and a6_offset to an initial value since they are used in
autorange fn. */
  a6_gain=ideal_A6_gain;
  a6_offset=0;

  maxx(0); select(LA,GD);
  base(0x1820); crb(4,1);
  span(10,UPO); vrd(0); lts_wait(2);
  meas(5,SYNC);
  if(alarm) auto_range_meas(10);
  a6_offset=(res/11.0);
  if((a6_offset<-0.0005)||(a6_offset>0.0005)|| alarm){
    success=FALSE;
  }

  crb(4,0); crb(5,1);
  vrd(0); maxd(0); lts_wait(2);
  null(3,NOSYNC);
  x=rdvo; vrd(1); lts_wait(2);
  diff(5,NOSYNC);
  if(alarm) auto_range_diff(1);
  a6_gain=res/(x-rdvo);
  if((a6_gain<(ideal_A6_gain*0.9))||(a6_gain>(ideal_A6_gain*1.1))|| alarm){
    success=FALSE;
  }
  if(!success){
    run_time_error_msg_to_user(line1,line2,line3,TRUE);
  }

  hdwclr();

  return(success);
}
/*************************************************************************/
void auto_range_meas()
{

/* This function is to be used only with the DAC Family Board. It may be
    adapted for used with other boards.  */

double rdac_volt,pvx,nvx;
double mult_fact;

  alarm=0;
  maxx(10); meas(1,NOSYNC);
  if(alarm) return;
  rdac_volt=res/a6_gain-a6_offset+rdvo;
  if(span_value==512){
    pvx=10;
    nvx=0;
  }
  else if(span_value==768){
    pvx=5;
    nvx=-5;
  }
  else if(span_value==0){
    pvx=20;
    nvx=0;
  }
  else if(span_value==256){
    pvx=10;
    nvx=-10;
  }
  if(rdac_volt>pvx) rdac_volt=pvx;
  if(rdac_volt<nvx) rdac_volt=nvx;
  vrd(rdac_volt);
  if(nin==0x4000)  mult_fact=0.2;                /* select(x,CUR) */
  else if((nin==0x8000)&&(pin!=256)) mult_fact=2; /* select(x,VLTG) & x!=TH
*/
  else mult_fact=1;                              /* all other selects */
  maxx(0);
```

```
  meas(1,NOSYNC);
  if(alarm){     /* if it alarmed with max=0, then autorange...*/
    do{     /* the 2nd number must be a tad larger or it doesn't converge */
      alarm=0;
      if     (max_value<(0.01*mult_fact)) maxx(0.011*mult_fact);
      else if(max_value<(0.02*mult_fact)) maxx(0.021*mult_fact);
      else if(max_value<(0.04*mult_fact)) maxx(0.041*mult_fact);
      else if(max_value<(0.18*mult_fact)) maxx(0.181*mult_fact);
      else if(max_value<(0.23*mult_fact)) maxx(0.231*mult_fact);
      else if(max_value<(0.61*mult_fact)) maxx(0.611*mult_fact);
      else if(max_value<(0.76*mult_fact)) maxx(0.761*mult_fact);
      else if(max_value<(1.73*mult_fact)) maxx(1.731*mult_fact);
      else if(max_value<(2.43*mult_fact)) maxx(2.431*mult_fact);
      else if(max_value<(4.10*mult_fact)) maxx(4.101*mult_fact);
      else if(max_value<10)       maxx(10);
      meas(1,NOSYNC);
    }while((alarm)&&(max_value!=10));
  }
  return;
}
/*************************************************************************/
void auto_range_diff()
{
/* This function may be used with any Family Board */

double mult_fact;

  alarm=0;
  maxd(10); diff(1,NOSYNC);
  if(alarm) return;
  if(nin==0x4000) mult_fact=0.2;                /* select x,cur */
  else if((nin==0x8000)&&(pin!=256)) mult_fact=2;/* select(x,VLTG) & x!=TH */
  else mult_fact=1;                              /* all other selects */
  maxd(0);
  diff(1,NOSYNC);
  if(alarm){     /* if it alarmed with maxd=0, then autorange...*/
    do{     /* the 2nd number must be a tad larger or it doesn't converge */
      alarm=0;
      if     (maxd_value<(0.01*mult_fact)) maxd(0.011*mult_fact);
      else if(maxd_value<(0.02*mult_fact)) maxd(0.021*mult_fact);
      else if(maxd_value<(0.04*mult_fact)) maxd(0.041*mult_fact);
      else if(maxd_value<(0.18*mult_fact)) maxd(0.181*mult_fact);
      else if(maxd_value<(0.23*mult_fact)) maxd(0.231*mult_fact);
      else if(maxd_value<(0.61*mult_fact)) maxd(0.611*mult_fact);
      else if(maxd_value<(0.76*mult_fact)) maxd(0.761*mult_fact);
      else if(maxd_value<(1.73*mult_fact)) maxd(1.731*mult_fact);
      else if(maxd_value<(2.43*mult_fact)) maxd(2.431*mult_fact);
      else if(maxd_value<(4.10*mult_fact)) maxd(4.101*mult_fact);
      else if(maxd_value<10)       maxd(10);
      diff(1,NOSYNC);
    }while((alarm)&&(maxd_value!=10));
  }
  return;
}
/*************************************************************************/
void user_before_testing_part()
{
   strcpy(product_name,PRODUCT_ID);  /* This must be here. Used by system */

   return;
}
/*************************************************************************/
void user_after_testing_part()
{

   return;
}
/*************************************************************************/
void define_bin_descriptions()
{
   /* only used bins need to be defined. Highest bin is 32 */
```

```
    bin_descriptions[1]="Grade A Part";
    bin_descriptions[2]="Grade B Part";
    bin_descriptions[11]="Supply Current ";
    bin_descriptions[13]="Full-scale Error";
    bin_descriptions[15]="Zero-scale Error";
    bin_descriptions[19]="Output Leakage Current";
    bin_descriptions[22]="Input Current";
    bin_descriptions[23]="Input Resistance";
    bin_descriptions[26]="INL";
    bin_descriptions[28]="DNL";
    bin_descriptions[30]="Power Supply Rejection Ratio";
    bin_descriptions[32]="Output Current";
    return;
}
```

Same Program Without a Limit Table

An equivalent version of the above test program, but with embedded limits instead of a limit table, has very few differences. Namely:

- No limit table would be required, of course
- At the end of every test, instead of the lines:

```
auto_test_parameters_and_bin(value);
if(check_fail()) return;
```

the following would be needed:

```
test_parameters(value,"Name of this test...","mV",2,4.50,5.50,28);
bin_current_test(value);
if(check_fail()) return;
```

Compiling the Test Program

Programs may be compiled from inside Microsoft's Visual Workbench environment or from the DOS command line. We recommend compiling and linking from the DOS command line because of its simplicity. Using the Visual Workbench from Windows may be more convenient, but setting all the necessary options for proper compilation can be a frustrating task, especially if switches are inadvertently modified.

A batch file has been provided for compiling from the DOS command line. It is called *compile.bat* and is run by simply typing:

compile <*program_name_prefix*>

where <*program_name_prefix*> is the name of the test program source file, without the ".c" extension. For example to compile the file DAC_1234.C, type:

compile DAC_1234

The *compile.bat* batch file contains the following commands and switches:

```
cl /c /AL /Gt /G2 /Z7 /f- %1.c
```

Where :

cl This is the name of the Microsoft C-compiler itself.

/c This compiler switch instructs **cl** to compile without linking yet.

/AL This compiler switch indicates that the Large memory model should be used when compiling. This point is very important.

/Gt This switch instructs the compiler to place data larger than 256 bytes in the far heap instead of the near heap. This is needed because of the large amount of global variables used in the system code.

/G2 This switch instructs the compiler to generate 80286 (and above) instructions. It results in more efficient object code.

/Z7 This compiler switch instructs **cl** to include symbolic information in the object code. Without this switch, Codeview will not be able to debug the program properly.

/f- This switch instructs the compiler to select the optimizing compiler instead of the fast compiler. It results in more efficient object code.

%1.c The "%1" will be replaced with the command-line argument following the word "compile" when the batch file is run. In other words, it will be replaced by the test program prefix. The ".c" is appended in the batch file; this is why the ".c" extension must not be typed in the test program name when running the batch file.

Before compiling, make sure that the proper environment variables have been set. This can be done by executing the Microsoft batch file *msvcvars.bat* . This batch file is located in the BIN sub-directory of the C-compiler directory and can be called in the autoexec.bat file for convenience.

Linking

Once the source file has been compiled, the resulting object file must be linked with the PCLTS.LIB system library and with any other libraries or user modules (like a GPIB board library, for example) to generate the final executable program.

A batch file has been provided with the system files which will allow the user to easily link from the DOS command line. It is called *linkit.bat* and is run by simply typing:

linkit <*program_name_prefix*>

where <*program_name_prefix*> is the name of the test program source file, without extension. For example to link the file DAC_1234 with the system library type:

linkit DAC_1234

The *linkit.bat* batch file contains the following commands and switches:

```
link /CO /ST:4000 /NOI %1,%1,,pclts.lib;
```

Where :

link This is the name of the Microsoft linker.

/CO This switch instructs **link** to preserve symbolic information so that the executable program may be debugged using Codeview.

/ST:4000 This sets the size of the run-time stack. If the resulting program generates a "*stack overflow*" run-time error, replace this number for a larger one. Be conservative, a number that is too large may generate a linker "*stack plus data exceeds 64*" error.

/NOI This switch instructs **link** to preserve case sensitivity in variable and function names. Microsoft C ignores case by default.

%1 The "%1" will be replaced with the command-line argument following the word "linkit" when the batch file is run. In other words, it will be replaced by the test program prefix.

When *linkit DAC_1234* is run, the resulting actual command will be

```
link /CO /ST:4000 /NOI DAC_1234,DAC_1234,,pclts.lib;
```

If a second user object module called, for example, *EXTRA_FNS.OBJ* and a GPIB module called, for example, *MCIB.OBJ* were to be linked with the test program, the command to type would be:

```
link /CO /ST:4000 /NOI
DAC_1234+EXTRA_FNS+MCIB.LIB,DAC_1234,,pclts.lib;
```

Please refer to Microsoft documentation for details on the usage of LINK.

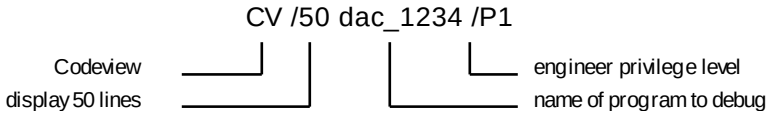
Debugging the Test Program

Using Microsoft's Codeview Debugger

Codeview is a debugger provided by Microsoft as part of the C compiler package. It is an excellent tool for program development and has all the necessary features for test program debugging.

To run it, simply select the ***Codeview Debugger*** option of the ***Development Menu*** in the LTSSHEL. Remember that in order to access this menu, the shell must be run using the "/P1" DOS command-line switch.

Of course, Codeview may also be run directly from the DOS command line. To do so type:



The /50 switch is optional, but is recommended. It allows the debugger screen to display 50 lines (VGA mode) instead of only 25, allowing the user to display a bigger portion of the test program and watch more variables. The /P1 switch is also optional but convenient.

Please consult the C compiler user's manual for full details on the entire set of Codeview command-line switches.

Running Codeview from the DOS command-line, instead of running it from LTSSHEL, is preferable if the system is low in available DOS memory.

Opening up the Necessary Windows

When Codeview is loaded for the first time, three windows will appear: the *locals* window, the *source* window and the *command* window. The *locals* window will display the current value of all the local variables in the function currently being executed. The *source* window displays the source code of the function currently being executed. And the *command* window enables the user to enter Codeview commands at any point in time.

We recommend that instead of the *locals* window, the *watch* window be used. The *watch* window is also used to display the current value of variables, but it will do so only for the variables that the user selects.

To close or open a window, use the ***Windows*** menu option. Windows may be moved and re-sized as needed. The recommended window setup is shown on the screen below.

Setting a Trap at the Start of *test_program()*

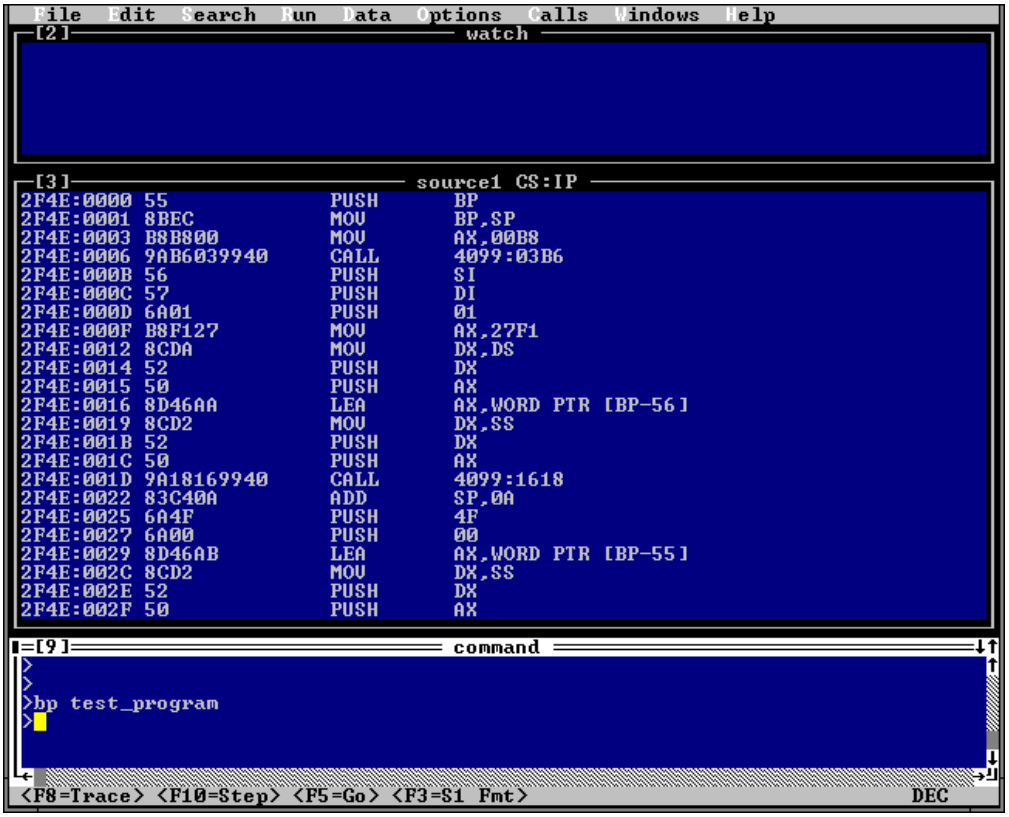
When Codeview is loaded, the *source* window will show the source code for the function currently being executed. The first function to be executed in every C program is the *main()* function, but in the case of the PCLTS system, *main()* is part of the library, so the source code for that module is not available. The initial source window will, for this reason, always show assembly language mnemonics which corresponds to *main()*.

The portion of the program where the user really wants to be is the *test_program()* function. The program will execute that function the moment the TEST key is hit, but the debugger will not stop at that function unless a trap, or *breakpoint*, as Microsoft calls it, is placed at that point.

To set a breakpoint at the start of the *test_program()* function, the user must type

```
bp test_program
```

at the *command* window prompt (see screen below).



Setting a Trap at Start of *test_program()*

Running the Program

Once a breakpoint at the *test_program()* function has been set, the program may be executed by using Codeview's ***Go*** command. This may be done by using the ***Run*** pull down menu, by hitting the F5 key, or by clicking on the <F5=Go> icon at the bottom of the screen. At this point the *main()* function will be running and the PCLTS test screen will appear, waiting for the user to make a menu selection or hit the TEST key (F1). When F1 is hit, *main()* will present the Operator Data Entry dialog window and the proceed to execute the *test_program()* function. If the breakpoint has been set, the program will stop at the beginning of the function (see screen below).

At this point the user has access to the source code, which can be scrolled to view and to set further breakpoints at any point in the code.

If no breakpoints have been set, other than the one at the start of *test_program(),* the test program will execute completely and return to the PCLTS' *main()* to wait for the use to hit F1 again or exit the program by hitting F8. If F8 is hit, *main()* will terminate and control will return to Codeview. At this point the entire program must be re-started by using Codeview's ***Run*** pull-down menu.

Trapping and Single-Stepping

It is very easy to set breakpoints in Codeview when the source code is on the screen: simply double-click on the line of code where the breakpoint is needed. To remove the breakpoint double-click again on the line. Breakpoints may also be set and edited using the ***Data*** pull-down menu.

Once further breakpoint have been set, the program may be executed up to the next breakpoint by using the ***Go*** key (F5).

To execute one line at a time (single-step) use the **Step** key (F10).

To single step through the code of a user function, instead of executing the entire function in one step, use the **Trace** key (F8) instead of F10. Once Codeview is inside the body of that function, F10 may be used again to step through the lines.

Conditional Breakpoints

Codeview allows setting breakpoints that will stop the execution of the program when a given variable changes value or becomes true, rather than at a specific line in the code. This may be done by using the ***Data-Set Breakpoint*** menu option.

Do Not Use *Trace* With System Functions

The F8 key (**Trace**) should only be used to trace user functions and not for system functions (like *meas()*, *vsa()*, *span()* etc.). If this is attempted, Codeview will go into the code for that function, but will display that code in assembly language since the source code is not available. It will then take a lot of single stepping through assembly code to find the end of that function.

A quick way to exit the function could be to hit F5, but unless there is a further breakpoint in the test program, the entire program will be executed to the end.

Watching Variables

The user may display the current value of any number of global and local variables in the *watch* window. Variables may be added and removed from the window by using the ***Data*** menu.

Formatting

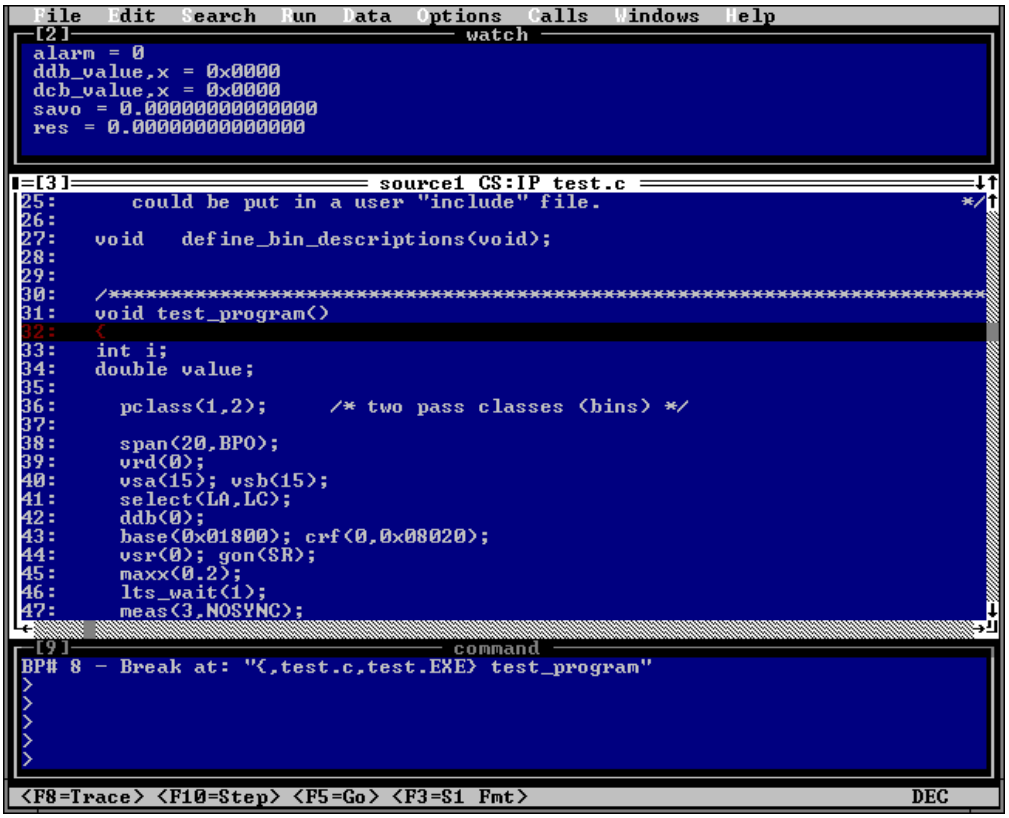
The display format of data being watched may be set by appending a formatting character to the variable (separated with a comma). For example to display the value of a variable in hexadecimal instead of decimal add an "x" to it:

ddb_value,x

To display a character string as an actual string (instead of the address (pointer) to the string) use an "s":

name,s

Consult Microsoft's documentation for other formatting characters.



Watching Variables in Codeview

Watching the Proper Variables

It is easy to get confused with what variables to display in the *watch* window when switching from the LTS Basic system to the PCLTS system. Here are two tips:

- *vsa* is a function to set source A. It is not the voltage that vsa was set to. The variable *vsa_value* is the one that has that voltage value, and *savo* has the actual voltage that the system could set in that source.

  If *vsa* is watched, instead of *vsa_value* or *savo*, Codeview will display the starting address of the *vsa()* function.

- *ddb* is a function to set the DDB lines. It does not keep the current value of the lines. The *ddb_value* variable is the one that does that.

  If *ddb* is watched, instead of *ddb_value,* Codeview will display the starting address of the *ddb()* function.

The case is analogous with *dcb()* and *dcb_value*, *maxx()* and *max_value*, *maxd()* and *maxd_value,* and others.

Executing Functions While Debugging

Just like with the LTS Basic interpreter, Codeview allows for executing functions while trapped at any point in the test program. To do so, simply type the name of the function with the corresponding arguments (or empty brackets if no arguments are required) at the *command* window prompt. The function name must be preceded by a question mark. For example, to modify the state of the *ddb* lines type:

```
?ddb(0x1ae5)
```

Or to take another measurement, type

```
?meas(1,NOSYNC)
```

Note that if the function being executed returns a value, the returned value will be displayed immediately after the function call in the *command* window. In the above example of taking a measurement, the value of the system variable *res* would be displayed (since *meas()* returns *res*).

Previously executed functions may be called again, without the need to re-type them, by scrolling down the *command* window, re-positioning the cursor at the desired function using the mouse, and then hitting the Enter key.

All of the functions that affect the hardware may be executed from the *command* window. However, some functions, like the ones related to the classification system (*aclass()*, *pclass()*, *fclass()*, *test()*, etc..) may not be executed while debugging the program.

The *command* window may also be used to assign new values to variables. To do so, type the assignment preceded by a question mark:

```
?value=3.2
```

Modifying Variables "On the Fly"

Variables may be modified while debugging in two different ways. If the variable to be modified is also being displayed in the *watch* window, then simply position the cursor by the current value being displayed and edit it on the *watch* screen.

The second way was discussed above and is to type the variable assignment at the command window preceded by a question mark.

Removing Screen Flicker

When a program is being debugged using Microsoft's Codeview debugger, there are two screens in two separate video pages: one is Codeview's user interface screen, where the program's source code can be seen, and the other is the screen of the program being debugged, in this case, the test program main screen. Every time the user single-steps through lines of code, Codeview momentarily switches to the test program's main screen and then back to Codeview's screen again, causing a rather annoying flicker. If the **Screen Swap** mode is turned off (using Codeview's Options menu), only Codeview's screen with the source code will be seen when single-stepping and the flickering will go away.

However, a confusing situation will arise when the test program is executed until the end. At the moment the *test_program()* function returns to the *main()* function (which is part of the system), the test program will be back to its main screen, waiting for the operator to either make a menu selection or hit TEST. If the Screen Swap mode was still on, the screen would show the test program's main screen at this point, but since it is turned off, the user will still see the Codeview screen, even though it is the test program that is in control. When this happens, the screen will appear to be in a "hung-up" state since it will not respond to the mouse. A good way to test if the test program is in control at that moment is to hit F7. If the Current Setup pop-up window appears, then the test program is in control and all the user needs to do to execute the *test_program()* function again is to hit F1 (TEST).

The above description of Codeview functions is by no means complete. For a complete description of the debugger, please refer to Microsoft's documentation.

Codeview's "Output Was Lost" Error Message

When the screen swap option is disabled (see **Removing Screen Flicker** section above) Codeview will display the following error message the first time the program is run:

Application output lost; screen exchange is off

This message is warning the user that text was printed on the PCLTS test screen (the output screen) but that is was not displayed because the screen swap option has been turned off. This is not a problem. This message should be ignored.

Restarting The Program

When the *test_program()* function terminates, control will return to the *main()* function, and the PCLTS main test screen will be waiting for the user to make a menu selection or run the test program function again. If the user hits F8 (Exit) at this point, then the entire program will terminate, returning control to Codeview. At this point, the user may not hit F5 (Go) again until he or she "restarts" the program. This is done by using Codeview's ***Run*** pull-down menu.

The CURRENT.STS file

Codeview saves the setup of a debugging session in a file called CURRENT.STS. This file can be found in the INIT sub-directory of the C-compiler directory, and it contains all the breakpoints, variables being "watched", the graphics mode used, and all the screen setup (like windows opened, size of windows, etc.).

So it is not necessary to set a breakpoint at the start of the *test_program()* function every time that the debugger is loaded because breakpoints are saved from the previous session. The setup will be lost, however, if a program with a different name is debugged, since the CURRENT.STS file is erased and re-written in when this happens.

HDWCLR.EXE and *hdwclr()*

If Codeview is exited during a debugging session while trapped in the middle of the test program, the test hardware and device under test will be left powered up. This may cause hot switching when unplugging the hardware or the part and damage may result. To solve this problem, a DOS command-line version of the *hdwclr()* function is provided. This is an executable program which will clear the LTS hardware settings. To execute simply type:

```
HDWCLR
```

from the DOS prompt.

An alternative way of clearing the hardware when exiting the program in the middle of the code would be to execute the *hdwclr()* function from the Codeview command window right before exiting. To do this type:

```
?hdwclr()
```

from the Codeview command window.

Most Common Mistake Using C

One of the most common mistakes of the first-time C-language user is to use the equal sign "=" in an if statement to check for equality:

```
if(number=2){
  ...
```

```
}
```

In C, there is a special equal sign to check for equality: the double equal sign "==". The regular equal sign is used exclusively for assignment. So in the above example, the expression inside the parenthesis will always have a true logical value since is an assignment. The result will be that the code inside the *if* will always be executed.

The correct way of checking for equality in an if statement is:

```
if(number==2){
  ...
}
```

Using the Proper Data Types

It is important to use the proper data type when declaring variables. In the following example, assume the variable *reference_voltage* is declared as an integer, and then later in the program, the measurement results in the *res* system variable being 10.052 volts:

```
int reference_voltage;

meas(5,NOSYNC);
reference_voltage=res;
```

Since *reference_voltage* is of type integer, instead of having the value 10.052, it will have a value of 10. If this measurement was to be used later in the program the results could be catastrophic.

Another important related point: a similar situation occurs when doing mathematical operations using constants, C will assume a type depending on how the number is written in the source program. Consider the following example:

255/256

The result of this operation is zero because 255 and 256 are assumed to be integers and not floating point numbers. The division of two integers results in an integer.

To actually perform the division, the numerator should be written as a floating point number:

255.0/256

The result of the above division is 0.9961.

Declaring Array Sizes

Arrays in C-language are defined differently than in Basic. In Basic, the number that appears in the DIM declaration represents the last index of the array, while in C-language, the number in the type declaration represents the size of the array. Both Basic and C start their array indexes at "0".

As an example, consider an array of integers named ABC and with a maximum size of five elements:

In Basic:	In C-language:
DIM ABC[4] !size of array is 5	**int abc[5]**; /* size of array is 5 */

Both arrays have the same range of indexes: abc[0] through abc[4], but they are declared differently.

The user must be very careful when declaring array sizes during a Basic-to-C conversion. All the array references inside the test program body may be left unchanged, but the declarations in C-language must be one number larger than their DIM Basic counterparts. Note that if the declaration is left the same, the C compiler will not reject it, but during run-time the array will overwrite memory space that has not been assigned to it by the system, often with catastrophic results. These type of problems are very hard to debug.

PCLTS User Interface

The LTSSHEL System Shell

The PCLTS system includes a central shell that acts as a "center of operations", from where all activities may be initiated. From the shell's main menu, a test program may be selected and loaded, a test development engineering session may be started, system parameters may be modified and datalog files may be processed.

After performing each task, the system always returns to the shell's main menu.

Privilege Levels

The shell is run from the DOS command line by typing:

LTSSHEL /P1

The **/P1** switch (privilege level 1) instructs the system to load under engineer privilege level. This option allows the user to have access to the ***Development Menu*** and also to the ***Engineering*** pull-down menu of the test program, both of which are inaccessible from the operator privilege level. If the **/P1** switch is not used, the shell will be loaded with operator privilege level (**/P0**) by default. In this way, sensitive actions, like bypassing the hourly system calibration and modifying test programs, can be restricted to the appropriate users.

The Main Menu

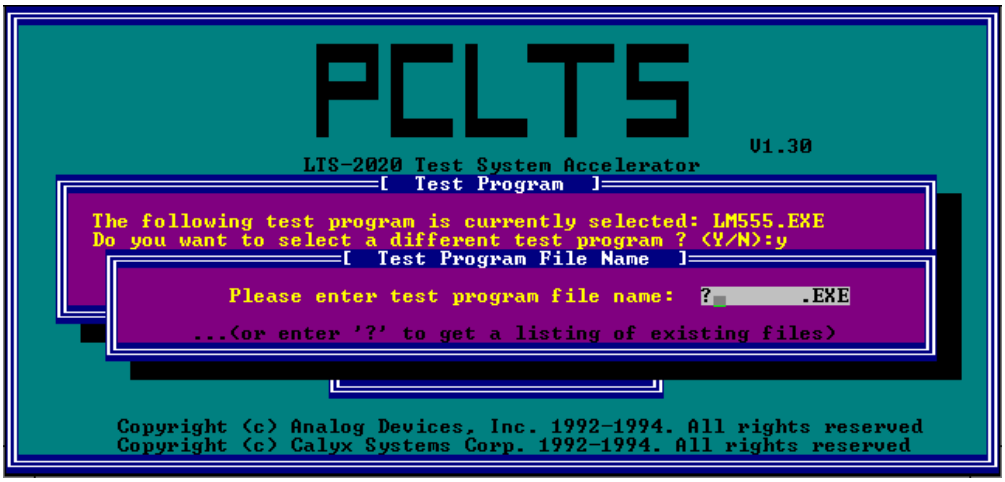
The LTSSHEL has a main menu with nine options which are selected by either entering the option number, or by using the arrow keys to move to the desired option and then hitting *Enter*. Some of the options will present a secondary pop-up menu.

The LTSSHEL main menu

Selecting the Test Program

Option number 1 on the main menu, ***Test Program Select***, selects the test program to be loaded. It does not actually load the program at that time (use option number 3 to load it). When this option is selected, a pop-up window will appear showing the test program that is currently selected, and asking the user if a different test program should be loaded. The system always remembers the name of the last test program that was selected by storing its name as part of the LTSSHEL.INI file.

If the user answers "Y" to selecting a different program, a second pop-up window will appear prompting for a test program file name. A "?" may be entered to view all the files with the ".EXE" extension that are available at the directory designated for the test programs (see **The LTSSHEL.INI Configuration File** section in the **Software Installation** chapter). Note that this file listing is only for viewing purposes, and the file may not be actually selected from that screen. The file name must still be typed in the prompt window.



Selecting a Test Program

Selecting a Handler Configuration File

Option 2 of the main menu, ***Handler Select***, allows the selection of a configuration file for the particular type of IC handler being used. Different handlers require different electronic interface parameters for proper operation (timing, voltage levels, binning, etc.).

A second menu appears, listing four popular commercial handlers, three custom options, and also the option of entering a different handler configuration file name. When a handler type is selected, the matching file must exist in the directory designated for handler configuration files (see **The LTSSHEL.INI Configuration File** section in the **Software Installation** chapter), and it must have the same name as the handler type but with the ".HND" extension.

Option 8, ***Filename***, is provided so that file names that do not appear in the menu may also be used. The file name prompt window offers the option of viewing all the files available in the designated directory.

Please refer to the **Generating Handler Configuration Files** section for details on generating and modifying handler configuration files.

Selecting a Handler Type

Loading the Test Program

Option 3 of the main menu is used for actually loading the selected test program. When this is done, the LTSSHEL screen menu is replaced by the test program screen with its pull-down menus. When the test program is terminated, the system returns to the LTSSHEL.

Verifying the Test Setup

The PCLTS system includes a hardware verification mechanism which can be exercised before testing any parts to ensure the integrity of the test hardware. This done by selecting option 4, ***Verify Setup***, on the main menu.

Please refer to the **Running Verify Setup** section of the **Other Utilities** chapter for more details.

Temporary Escape to DOS

Option 5 of the main menu, ***DOS Shell***, initiates a temporary DOS prompt session. This option may be used for performing DOS prompt tasks such as printing files, editing files, etc. After the task is done, the user must type EXIT to return to the LTSSHEL (which is still resident in memory).

It is important to note that under no circumstances should the user type LTSSHEL to return to the shell. If this is done, the user will not be really returning to the resident copy of LTSSHEL, but rather a second copy of the program will be loaded in

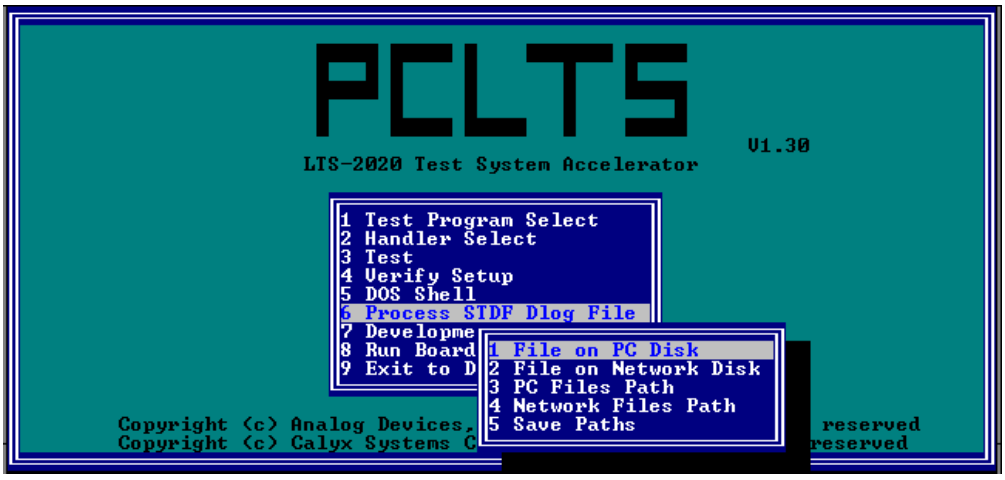
memory, causing the amount of available RAM to be significantly reduced. This condition will eventually cause an "insufficient memory" error message while the test program is being run.

If the user wants to permanently exit the LTSSHEL, option 9, ***Exit to DOS***, should be used.

Processing a Datalog File

The PCLTS system stores datalog data using binary files that adhere to the STDF industry standard. This allows for more compact files that contain more information, including an embedded summary sheet. Because these files are binary, as opposed to ASCII text, they may not be directly printed or viewed. Option 6 of the main menu, ***Process STDF Dlog File*** should be used to extract information from datalog files.

The user must select the location of the datalog file. It may reside either on the PC's hard disk or on the network server disk (options 1 and 2 of the secondary pop-up menu). The file path within the PC's disk or the server's disk can be modified using options 3 and 4 of this menu. These paths are part of the LTSSHEL.INI system configuration file and indicate the directory to where datalog files are sent during test program execution. If the paths are modified from this menu, the changes will be temporary and will last only during the current file processing. They may be permanently modified, however, by using option 5, ***Save Paths***. This will update the LTSSHEL.INI file to the newly selected path names.



Selecting the Location of a Datalog File

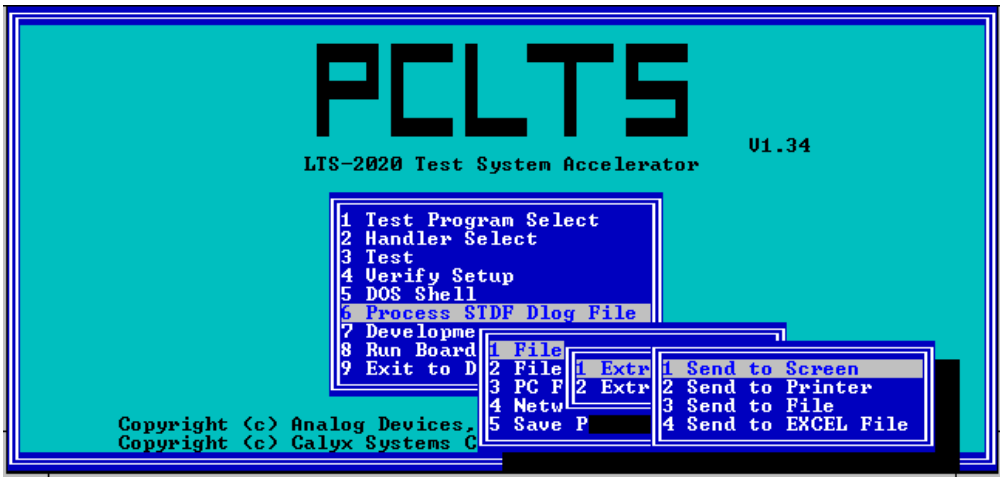
Entering a New Datalog Path Name

A third child pop-up menu appears once the location of the datalog file has been selected. This menu has two options: ***Extract Datalog*** and ***Extract Summary***. A fourth menu is used to select any of four destinations for the extracted datalog or summary sheet: screen, printer, ASCII file, or EXCEL text file. The ASCII file option creates an ASCII file containing all the exact characters that are displayed on the screen when the system datalogs to the screen.

The EXCEL file option also creates an ASCII file but in the Microsoft EXCEL text format. Note that this is not a "**.XLS**"-type binary file. To read it into EXCEL you must import it as a text file. This file may also be read by most other Windows spreadsheet applications.

If the screen is chosen, the extracted information will scroll through the screen. While it is doing so, the user may hit the *Pause* key to freeze the scrolling and then any other key to continue. Once all the information has been displayed on the screen, the *Page Up* and *Page Down* keys may be used to scroll back up and down. The *Home* and *End* keys will display the top and bottom of the scroll buffer, respectively. Note that the scroll buffer has a finite size and is eventually filled. If this happens, the information first displayed will be lost from the scroll buffer and cannot be displayed with the *Home* key.

For long datalog files, it is more helpful to send the extracted information to an ASCII file and then view this file with a text editor. Note that if an ASCII file is chosen as the target, it will always have the ".TXT" extension. It will reside in the same directory as the STDF source file.



Selecting Destination of Information Extracted from STDF File

Board Diagnostics

Option 8 of the LTSSHEL main menu, ***Run Board Diagnostics***, executes the hardware diagnostics utility. The purpose of this utility is to diagnose malfunctions of the CPU Emulator and the Bus Driver boards and also to ensure the integrity of the interconnecting cables. For details on running this utility please refer to the **Troubleshooting Hardware** chapter of this manual.

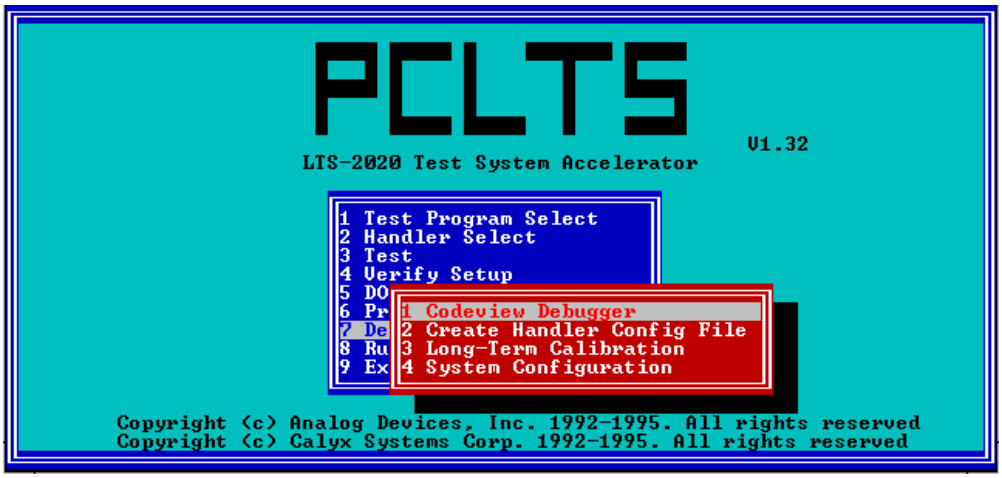
Exiting the Shell

Option 9 of the LTSSHEL main menu, ***Exit to DOS***, terminates the LTSSHEL program.

The Development Menu

This menu accesses tasks typically performed by the test development engineer and is available only at the engineer privilege level (level 1). To load the shell using this privilege level the /P1 switch must be typed at the DOS command line:

LTSSHEL /P1



The Development Menu

Codeview Debugger

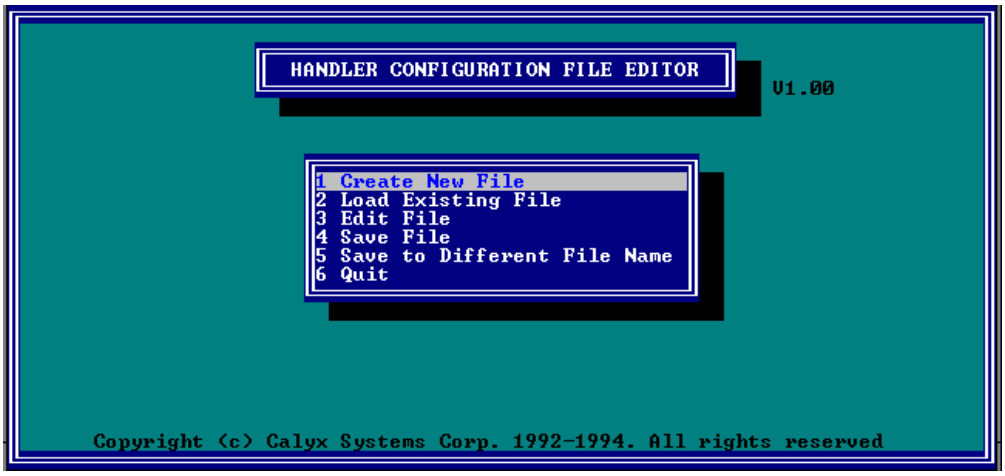
Option 1 of the Development Menu loads Microsoft's Codeview debugger. It passes the test program executable file name (".EXE" extension) as an argument together with the privilege level being used.

For more details please refer to Microsoft's documentation and the **Test Program Debugging** section.

Creating Handler Configuration Files

Option 3 of the Development Menu enables the user to edit and create handler configuration files. These files contain timing, voltage level and binning information associated with a particular handler. These files are loaded in memory by the test program and are used to control the electrical interface with the handler. Consult your handler's technical manual for details on its interface parameters.

Parameters are entered by means of a "form" which is presented to the user when option 1 of the Handler Configuration File Editor is selected. It is easier, however, to use option 2 and load an existing configuration file which can be modified and saved under a different name.



Handler Configuration File Editor Main Menu

The form is easily filled by moving the cursor with the arrow keys and the Enter key. When the last field is reached, the system prompts the user to hit Enter again to accept all the current entries or any other key (probably an arrow key) to edit any of the entries.

The following is the format for the parameters:

Logic Levels of Lines

These five entries set the digital logic levels (high or low) of the handler port signals. They must be either be **L** or **H**, for low and high, respectively.

Use RETEST IN as EOW

A "**Y**" in this field instructs the system to use the Retest In line (sent from the handler to the tester) as an **E**nd **O**f **W**afer signal instead. The EOW signal is sent by some modern wafer probers (like the KLA automated probers) to indicate that the current wafer has been finished and that a new wafer is about to be automatically loaded. On detection of this signal, the PCLTS system will do the following tasks:

- Send the current summary sheet to the printer
- Close any open datalog files
- Clear the summary information and prepare for a new lot

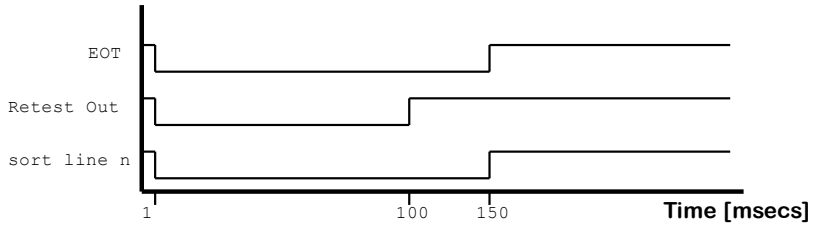
This allows for continuous unassisted sorting on an entire wafer "boat".

Max Voltage of START TEST Line

This value is the maximum voltage expected on the handler's Start Test line logic high level. It is used by the system to program the threshold DAC on the LTS console's IEEE/Handler board.

Ton and Toff Time Points

These are time points, in milliseconds, at which the End of Test, sort lines and Retest Out signals should occur. They determine the timing of the handler interface. The following time line represents the timing selected on the form in the figure below:



Handler Interface Timing Diagram (matches form in screen below)

Bins Assigned to Sort n Line

The LTS-2020 IEEE/Handler Board supports 8 hardware sort lines which connect to handler via the handler port. Whenever a part is done testing, the End Of Test (EOT) signal is pulsed together with one of the 8 sort lines. This lets the handler know on what rail it should drop the part.

The system software, on the other hand, may catalog a part as falling into any of 32 possible bins. These 32 bins must then be mapped to the 8 physical sort lines that connect the tester with the handler. This is done by assigning one or more software bins to the eight sort lines.

To assign a bin, enter the bin number in the allocated space in the form. More that one bin can be assigned to a particular sort line. If this is the case, the different bins should be separated by commas. Ranges may also be assigned by separating the starting and ending bins in the range with a dash (**-**) or the word **TO**. An asterisk (*****) should be entered if no bins are assigned to that particular sort line.

Example:

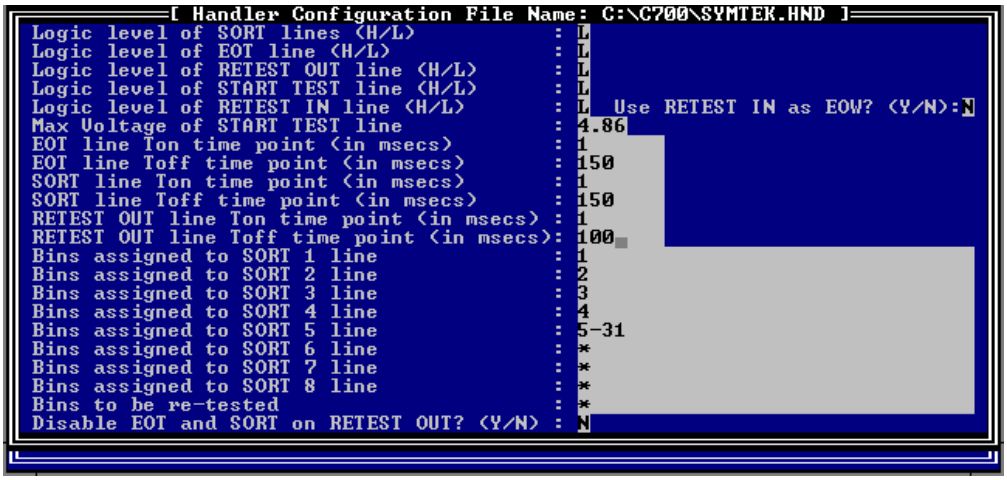
1,4,7-10 or 1,4,7TO10	===>	bins 1,4,7,8,9 and 10 are assigned to that sort line.

Bins to be Retested

If a device results in any of the software bins listed in this entry, the system will pulse the End Of Test signal. The timing will depend on the Ton and Toff timing parameters set. An asterisk (*****) should be entered if no bins are selected.

Disable EOT and SORT on RETEST OUT

As explained above, if a given bin, *n*, has been set as a bin to retest, and the current part results in that bin *n*, then the Retest Out line will be pulsed. By default, the system will also pulse the End of Test and the resulting sort line when this happens. However, by entering "**Y**" in this field of the form, the user has the option disabling the pulsing of these lines when a Retest Out is sent.



Modifying Handler Configuration File Parameters

Special handler configuration files can be created for isolating parts corresponding to a specific type of fail bin. For example, if with a particular lot of parts, the user wants to perform bench measurements on all bin 25 devices, he or she can isolate them in the handler by assigning only bin 25 to sort line 7. This special configuration file can have a distinct file name which can be then loaded using the ***Filename*** option of the ***Handler Select*** option in the LTSSHEL main menu.

Long Term Calibration

This option runs the long-term calibration program which is part of the semi-annual recommended LTS-2020 maintenance procedure. It should not be confused with the *hourly* system calibration.

When this option is selected, the system launches the LONGCAL.EXE program. LONGCAL behaves just like a regular test program, and is an identical clone of the LTS Basic calibration program. It supports using both an HP-3656 or an HP-3458 digital voltmeters.

Summary sheets and datalogs can be sent to the screen and/or printer in the standard way.

System Configuration

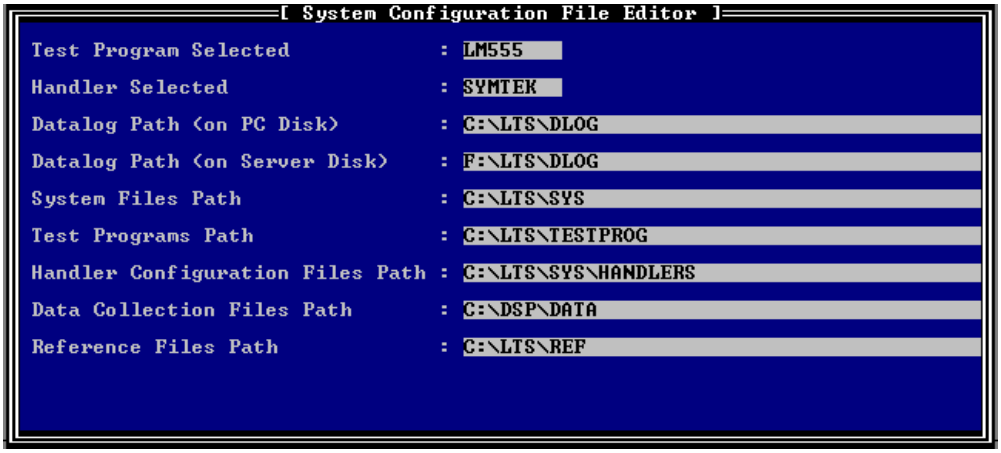
The PCLTS system stores path information and other configuration parameters in a system file called LTSSHEL.INI. This file can be modified by using the System Configuration Editor accessed through option 5, ***System Configuration***, of the Development Menu. After modifying the file, it should be saved for the changes to take effect. Some parameters the are part of the LTSSHEL.INI file, like the time of

the last successful hourly calibration, are not to be modified by the user and therefore do not show up on this editor.

System Configuration Editor Main Menu

Some of the system parameters that appear on the edit form, like the test program name, the handler selected and the datalog paths, can also be modified from other menus in the system.

The form is filled by moving the cursor with the arrow keys and the Enter key. When the last field is reached, the system prompts the user to hit Enter again to accept all the current entries or any other key (probably an arrow key) to edit any of the entries.



Modifying System Configuration Parameters

The Test Program Pull-down Menus

Test programs have a wide variety of options which can be selected by means of pull-down menus. The menus are activated by using the slash key ("***/***"). When this is done, the top bar on the screen will become highlighted to indicate that the menu is activated and ready for a selection. Menu options on the horizontal bar can be then selected by either using the horizontal arrow keys ("←" and "→") or by typing the first letter of the word identifying the option desired. For example, to pull down the ***Summary*** menu, hit "***/***" followed by "**S**". Options within the vertical pull-down menus may be selected by either using the vertical arrow keys ("↑" and "↓") or by typing the number identifying the option desired.

Once the selection is completed, the pull-down menu bar will automatically de-activate itself. To de-activate the pull-down menu bar before a selection is completed, simply hit the "**Esc**" key.

Note that the function keys may not be used and the test program may not be run while the menus are active.

THE DATALOG MENU

Data may be logged to the screen, printer, local disk, or network server disk. This may be done in any combination whatsoever.

Note that In the PCLTS system data is collected in memory while the test program is being executed, and is not "dumped" to the designated targets until the program has finished execution. This results in consistent test program timing, regardless of the datalog option being off or on.

Datalogging to the Screen

The most obvious target for logging test data is the screen. This may be done either by selecting the Dlog to Screen option of the pull-down menu, or simply by hitting the "**F5**" function key. The F5 key acts as a quick toggle key: hitting it will alternatively turn the Dlog to Screen option on and off.

If the tests in the program cover more than one full screen, it is almost impossible to watch test results as they scroll through the screen. The ***Screen Dlog Speed*** option to slow down the screen is provided for this reason.

In conjunction with the scrolling speed control, the "**Pause**" key and "**Page Up**" keys may be used to stop and continue the scrolling, respectively.

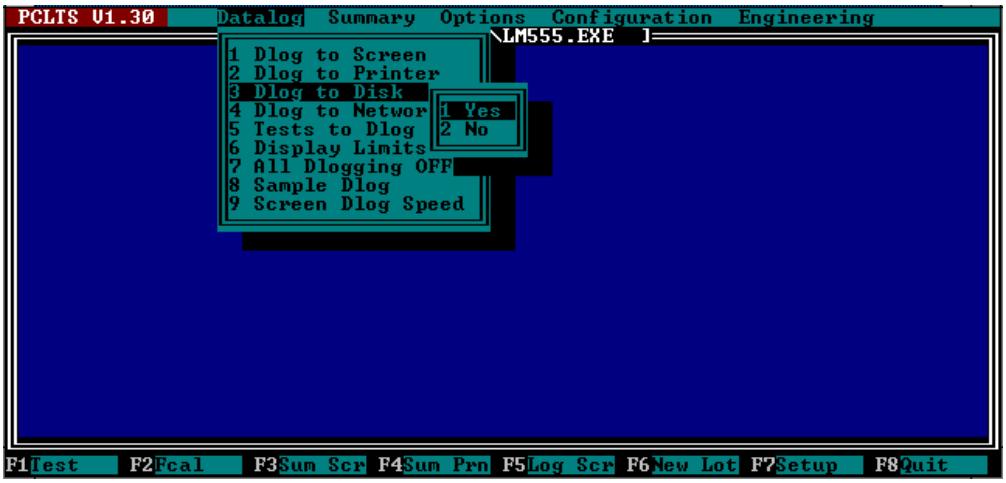
Interrupting Datalog to Screen

Sometimes it is desirable to interrupt the logging of the data to the screen before all tests have been displayed, especially when there is a very large number of tests and the Dlog Speed has been set to slow. If the **Ctrl** key is held pressed and the **Break** key (also labeled **Pause**) is hit at the same time, the datalog will be aborted. If this is done, the results for the current part will still be recorded in the summary sheet information and will be part of the lot statistics. Also, if datalogging to the disk was on, the data for the current part would still be written to the file. Only the display of the data on the screen would be aborted.

If **Ctrl-Break** is hit while the test program is actually running, then the testing of the current part would be aborted and no data or results would be recorded. It would be equivalent to not having tested the device at all.

Datalogging to Disk (Local or Network)

When ***Dlog to Disk*** or ***Dlog to Network*** is selected, the system prompts for a file name. The user has the option of entering "**?**" to view the existing datalog files found in the designated datalog directory. If the file selected already exists the system will issue a warning indicating that the existing file may be erased and replaced with the new data. Appending new data to a previously closed datalog file is not supported.



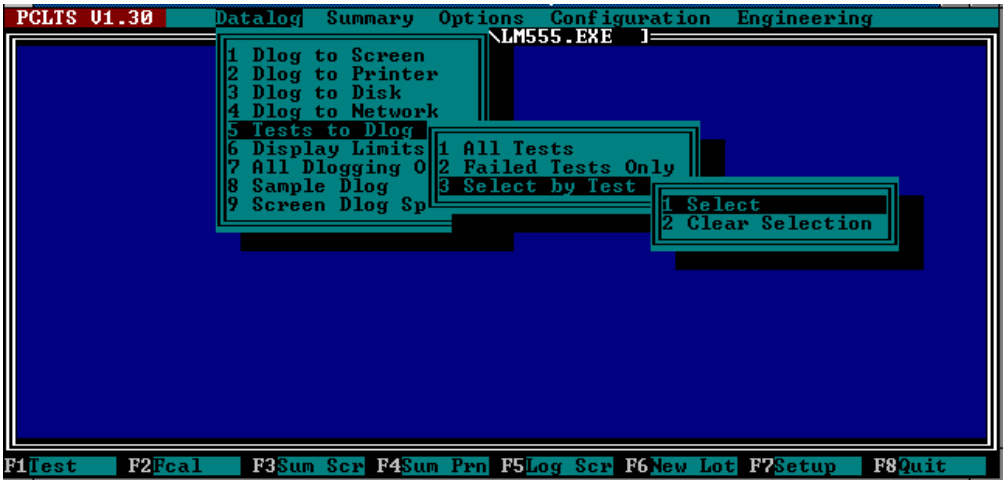
The Datalog Menu

Selecting Tests to Datalog

The user has the option of logging only a selected subset of tests by selecting the ***Tests to Dlog*** option. These may be either failing tests or a group of tests selected by test number.

If the **Failed Tests Only** option is selected, the user is warned that unless the program is forced to continue after the first failure (by using the ***Options*** menu), only the first failed test will be logged.

When selecting tests by test number, with the ***Select by Test*** menu option, up to 200 tests may be entered in any order. Tests that are entered twice are automatically corrected by the system. New tests may be added to the list of previously selected tests at any time. When this is done, the previous list will be displayed sorted.



Selecting Tests to Datalog

Displaying Test Limits

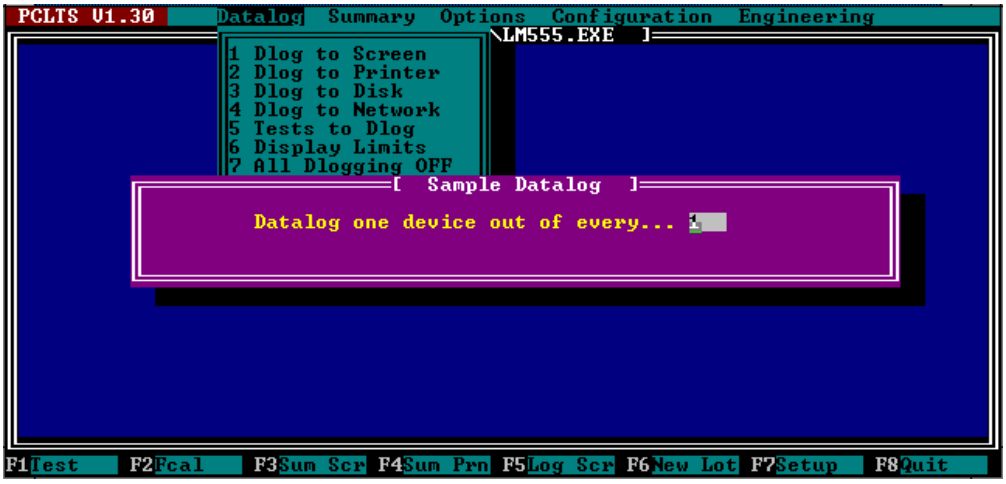
Tests limits may be optionally displayed when logging data to the screen or printer. When this is done, by means of the ***Display Limits*** menu option, the minimum and maximum limits will be displayed to the right of the test name.

Note that only the worst-case passing bin limits will be displayed. For example, if a given test has three passing bins, bin 1,2 and 3, only the bin 3 limits will be displayed.

Sample Datalog

When a very large number of devices will be tested, it is sometimes desirable to log only a sample instead of every single device. A good example of this is evaluating the yield of a wafer particular lot. Collecting data for every die of every wafer in the lot would result in an enormous amount of data.

By using the ***Sample Dlog*** menu option, the user has the choice of datalogging only one device out of every **x** devices, where **x** can be any positive integer number.



Sample Datalog

THE SUMMARY MENU

The ***Summary*** menu is used to send a summary sheet to either the printer or the screen. If it is sent to the screen, a magenta pop-up window will appear and the summary sheet will be displayed on it. If it covers more than one screen in length, the "**Page Up**" and "**Page Down**" keys may be used to scroll up and down through it. The "**Home**" and "**End**" keys may also be used to quickly go to the top or bottom of the sheet, respectively.

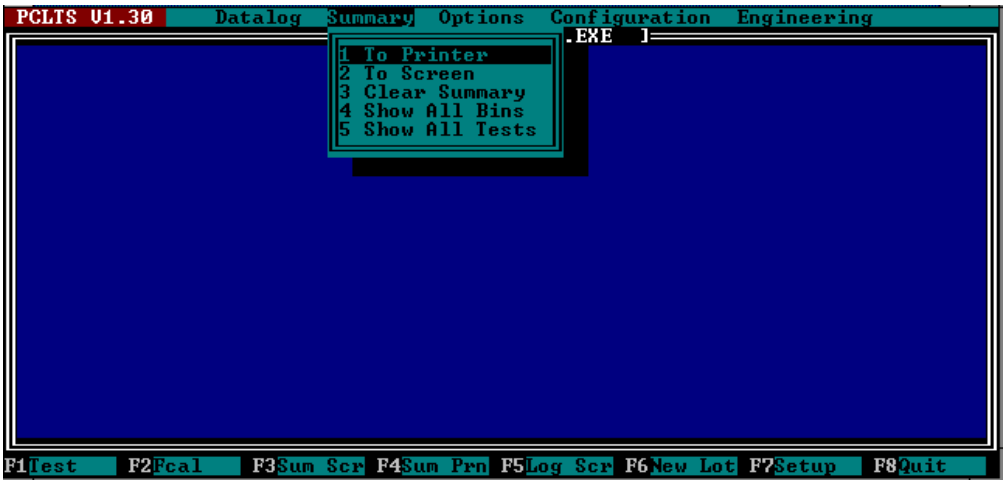
The summary sheet has three tables in it. The "Device Distribution by Bin" table shows all the bins in which the tested devices resulted, and how many devices resulted in each bin. The "Device Failure Count by Test" table shows all the tests that were failed with the number of devices that failed each one of the tests. Finally, the "Device Downgrade Count by Test" table shows the devices that passed, but

resulted in inferior passing bins (bins 2,3...). These tables show only the tests that were failed and only the bins with parts in them. In order to see a listing of all bins (even if there are no parts "in" them), and of all tests (even the ones that did not fail), the ***Show all Bins*** and ***Show all Tests*** menu options can be used, respectively.

After the third table, the summary sheet shows the average test time for both passing and failing devices. With both these numbers, together with the yield, can help the user estimate the total test time for the lot. This information can be very useful for production planning and scheduling.

Clearing the Summary Data

When the ***Clear Summary*** menu option is exercised, all the lot statistics are cleared from memory and the *snum* system variable is set back to 1. Before starting to test another parts lot, it is more convenient to use the ***New Lot*** option of the ***Options*** menu, which will, among other tasks also clear the summary data.



The Summary Menu

THE OPTIONS MENU

The ***Options*** menu contains nine functions that have been provided for user convenience. These are the following:

Start New Lot

When this option is selected the system will clear the summary and all the statistics related to the current lot of parts being tested. The *snum* variable is set to 1. All this is also done when clearing the summary, but the main difference with ***Start New Lot*** is that in addition to performing these functions, it will also close any open datalog files and clear all system variables. In fact, it is no different than exiting the program and loading it again. After starting a new lot, the Operator Data Entry dialog box will be again presented to the operator before testing the first device of the new lot.

Note that this option may also be quickly selected by hitting the "**F6**" function key.

Forcing Through Failures

The ***Force Through*** option is used to continue executing tests in the test program even after a test has already failed. By default, the PCLTS system will stop after the first failing test in the program.

Note that in the PCLTS system, when a part fails, the resulting bin for that device will correspond to the first failing test regardless of the program being in force-through mode or not. The original LTS-2020 ADOS system does not work in this fashion. For example: assume a device fails test number 5, with a fail bin of 21 and also fails test 10, with a fail bin of 24. In the PCLTS system, this device will result in a "Fail bin 21" part, even if the program is "forced-through" failures, since test 5 is the first test to fail. In the ADOS system, however, the device will result in a bin 5 part if the program is set to stop after the first failure and will, on the other hand, result in a bin 24 part if the program is set to "force-through" failures.

Retest

The system only supports re-testing the last device tested. When ***Retest*** is selected, the system will clear all the data from related to the previous device from the lot statistics and will prepare to test the next device, just as if that next device was being tested for the first time. The message "****DEVICE WAS RE-TESTED****" will appear on the screen after testing. If the system is logging data to a file, both sets of data will be logged, but the second will be marked as being a re-tested part.

This option is meant to be used when hand testing devices or when running a handler in manual mode.

Setting the Serial Number

The serial number of the next device to be tested may be modified by using the ***Serial Number*** option. This will only modify the *snum* variable, and will in no way affect the lot statistics. If the serial number is set to the previous device's serial number, it will not perform a true retest since the results from the previous device will not be deleted from the system.

Displaying the Test Time

When the ***Show Test Time*** option is set, the resulting test time, in seconds, will be displayed immediately after testing the part. Note that the test time of the first device tested is always longer than the true test time due to initial system "housekeeping". The average test time for a lot can also be obtained from the summary sheet.

Tone

An audible tone may be set to sound when a device results in a passing part, a failing part, or a particular bin number part. For example this can be used to alert an operator of an excessive number of bin 17 parts, which could be caused by a setup problem.

User Switches

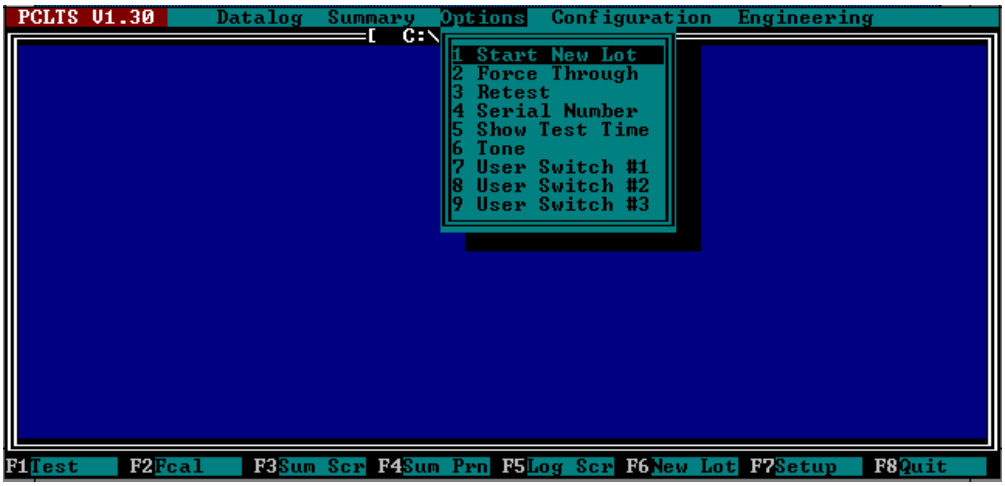
The test engineer has the option of using three different software switches to produce three variations on a single test program. When the operator sets one of these switches, a system software variable is set to a TRUE Boolean logic value. This variable may be used in the program to conditionally execute portions of the code of set a particular test condition. The three system software variables are *user_switch_1*, *user_switch_2*, *and user_switch_3*, all of the integer type.

As an example if the test program contains the following code at the top:

```
if(user_switch_2) vsa(10);
else vsa(5);
```

then, if the operator sets the ***User Switch 2*** using the menu, the test program will be executed with source A set to 10 volts instead of 5 volts.

All three switches are initially set to a FALSE Boolean logic value by default.



Options Menu

THE CONFIGURATION MENU

Handler Port

This option allows the handler port on the LTS-2020 console to be disabled. It is useful when the operator wants to interrupt the handler testing to connect a hand socket and test a device without the handler dropping its next part. Disabling the handler port will cause the system not to respond to the *Start Test* signal and the tester not to generate the *End Of Test* and sort line signals.

Family Board

This menu option will read the family board ID of the board currently plugged in the tester. If no board is plugged in, it will read 255.

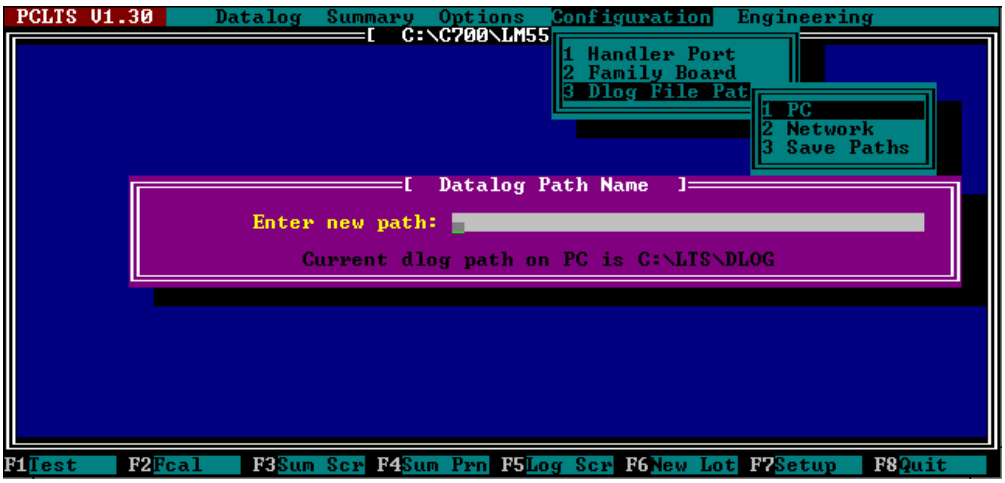
This ID is a hardwired eight-bit code that all family boards have in their circuitry. It can be read through the console's CRU I/O bus.

Dlog File Path

This option allows the operator to change the path designated for storing datalog files in both the PC's disk and in the network server disk (if connected to a network). These paths are stored in the LTSSHEL.INI file and are also modifiable from the LTSSHEL menu.

Once the user selects if the ***PC*** or ***Network*** path should be modified, the system will prompt for the new path name. At the bottom of the dialog window, the currently selected path will be displayed.

Any modification will last only while the test program is loaded, unless the ***Save Paths*** option is selected, in which case they will remain even after exiting the test program.



Modifying Datalog Paths

THE ENGINEERING MENU

The engineering menu can only be accessed if the program has been loaded under privilege level 1 (engineer privilege level). The reason for this is that this menu contains sensitive options such as bypassing the hourly measurement system calibration and resetting the system. To load the program with this privilege level add the "**/p1**" switch at the DOS command line.

Bypass Cal

Just like the original LTS-2020 ADOS, the PCLTS system calibrates the system's measurement board, voltage source board and digital I/O board, on an hourly basis. The user has the option of by-passing this calibration by using the ***Bypass Cal*** menu option. When a new test program is loaded, the system will again calibrate itself every hour.

Note that by-passing system calibration may result in erroneous measurements.

Collect Data

This option should not be confused with recording datalog data (see ***Datalog Menu*** section).

A function called *record_data_point()* has been provided in the PCLTS.LIB library which can be called from the test program in a section where data points want to be

recorded to a file for later processing. The function call in the program has no effect until the ***Collect Data*** menu option is set. When the operator activates data collection, the system will prompt for a filename. This file is an ASCII file with the "**.DTA**" extension. All points recorded will be followed by a carriage return and a line feed, resulting in the file having one data point per line.

Once data collection is selected (enabled), all calls to *record_data_point()* will result in one more point being added to the data file. When the next part is tested, the new data points will be appended to the file after previous part's data. The file may be closed at any point by using the ***Collect Data*** menu and selecting the ***Off*** option.

When the system prompts for the data collection filename, the user has the option of entering "**?**" to view the .DTA files that already exist in the directory designated for data collection files (the directory may be modified through the ***System Configuration*** option of the ***Development*** menu in the LTSSHEL). The system will warn the user if the filename entered in the dialog box already exists, and will prompt for an "**A**", an "**E**" or an "**N**" which represent the following options:

A Append the new data to the existing data in the file.

E Erase the existing file first and open again to record the new data.

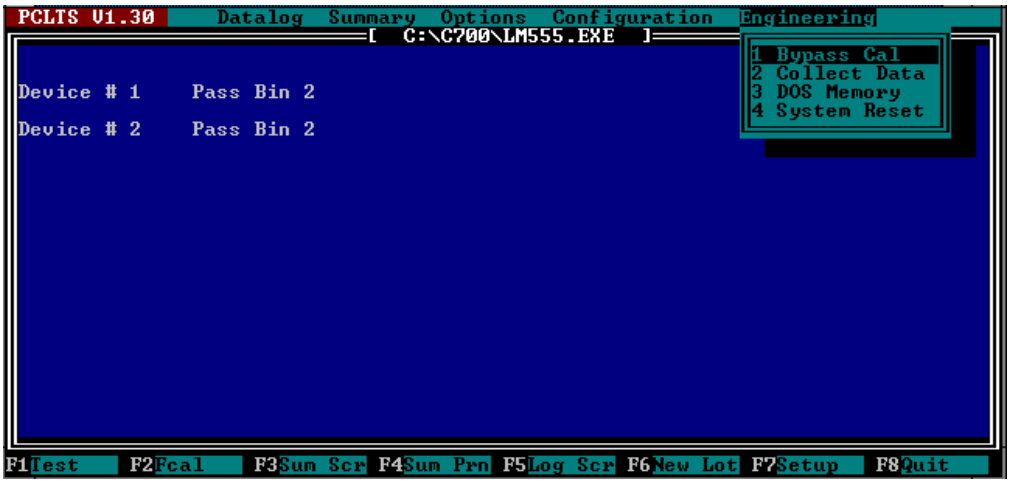
N Leave the existing file untouched and go back to the dialog box to enter a new filename.

System Reset

When the ***System Reset*** option is selected, the system will generate a 100 ms low pulse on the /IORST line of the LTS-2020 backplane. This will cause all the system boards to be reset.

The difference between this hard reset and calling the *hdwclr()* function, is that the hard reset will also clear the IEEE/handler board registers. These registers contain handler interface configuration information and if cleared the handler port will become inoperative, or at least erratic, until the test program is loaded again.

The *hdwclr()* function clears only the boards plugged into the slots in front of the bus driver board (voltage source, digital I/O and measure boards). It does so by actually writing zeros to all the registers rather than by pulsing the /IORST line like ***System Reset*** .



Engineering Menu

THE FUNCTION KEYS

The bottom of the test screen shows eight small rectangles which represent function keys. "**F1**" through "**F8**" on the keyboard are used to easily and quickly perform the most frequent operator tasks using just one key.

These keys may be hit at any point in time. If they are hit while the test program is running, they will take effect as soon as the program is finished.

F1 - Test

Starts execution of the test program. Is exactly the same as hitting the **TEST** button on the LTS-2020 console, and equivalent to the handler asserting the **Start Test** signal.

F2 - Force Calibration

Will cause the system to calibrate its measurement board, voltage source board and digital I/O board. It will then not calibrate for another full hour.

Hitting this key is equivalent to typing the FCAL statement in the ADOS LTS system.

F3 - Send Summary to the Screen

This is a short-cut to using the ***Summary*** menu to display the summary sheet on the screen. The result is identical.

F4 - Send Summary to the Printer

This is a short-cut to using the ***Summary*** menu to send the summary sheet to the printer. The result is identical.

F5 - Datalog to the Screen

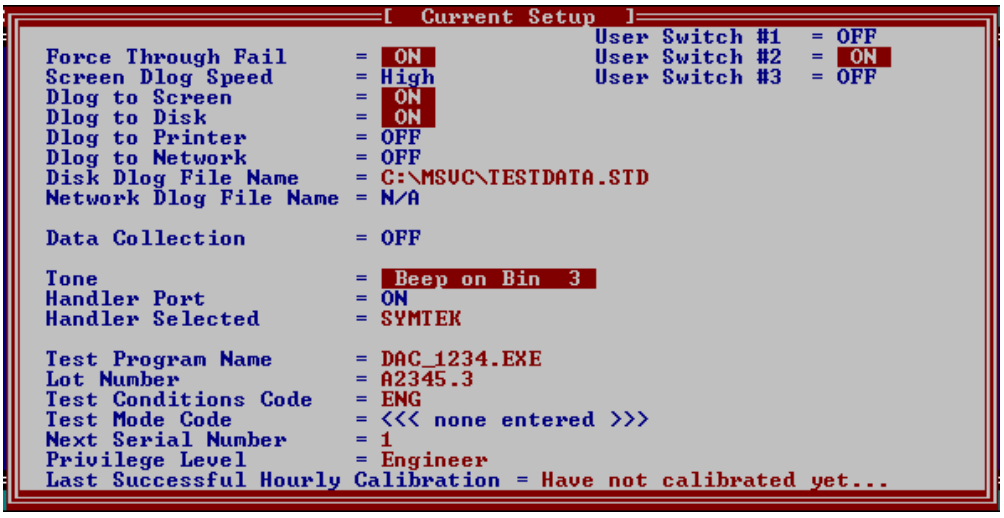
This is a short-cut to using the ***Datalog*** menu to log data to the screen. The result is identical.

F6 - Start New Lot

This is a short-cut to using the ***Options*** menu to start a new lot. The result is identical.

F7 - Display Current Setup

Hitting the **"F7"** key causes a window with the current setup to pop-up on the screen. This window displays the options that have been set by the user up to that point. Default settings will appear in blue and modified settings will appear in red.



Current Setup Pop-up Window

F8 - Quit the Test Program

The "**F8**" key is used for exiting the test program. If test results are still in memory, or if any datalog files are open, a warning window will pop-up and alert the user.

Using *Ctrl-Break* to Abort

The execution of the test program may be interrupted at any time by holding down the **Ctrl** key and hitting the **Break** key (also labeled **Pause**) at the same time. If this is done, no results will be recorded for the current device.

This same key sequence may be used to abort a datalog to the screen while the data is scrolling. If this is done, the results for the current part will still be recorded in the summary sheet information and will be part of the lot statistics. Also, if datalogging to the disk was on, the data for the current part would still be written to the file. Only the display of the data on the screen would be aborted.

Optimizing The Program

Producing a Well-Structured Test Program

The C programming language includes instructions like **while**, **do-while**, **for** and **break**, which enables the programmer to write code that flows logically from one section to the next, without "jumping" from one portion of the code to another. The use of this practice, known as structured programming, results in programs that are easy to follow and understand, and therefore maintain. Basic (especially older versions like LTS Basic) does not have the proper constructs to enable structured programming.

Avoid using *goto*

Most hard-to-read programs owe their lack of structure to the generous use of **goto**s. Even though C does include the **goto** instruction, it has a variety of other instructions which enable writing programs without *goto*s in them.

If the CONVERT program is used to convert a Basic program to C-language code, the resulting file will probably include several **goto** instructions (as many as there were in the original Basic program). It is highly advisable to edit the resulting C program and replace **goto** occurrences with alternative constructs.

Use Meaningful Variable Names

Variable names in LTS Basic are restricted to only three characters. This results in cryptic names that do not fully describe their use. C-language allows up to 31 characters (variable names may be longer, but characters past the 31st are ignored) which enables the use of highly descriptive variable names. The underscore character may be used to separate words within a given variable name. For example:

converter_output_voltage

Remember that C is case-sensitive, and therefore *converter_output_voltage* is different than *Converter_Output_Voltage.*

Divide and Conquer

It is always better when approaching a large and complex problem to divide it into several parts and solve each one independently. This is particularly true of a test program. Tasks should be logically divided into modules and each module implemented by means of a function.

Use Local and Global Variables

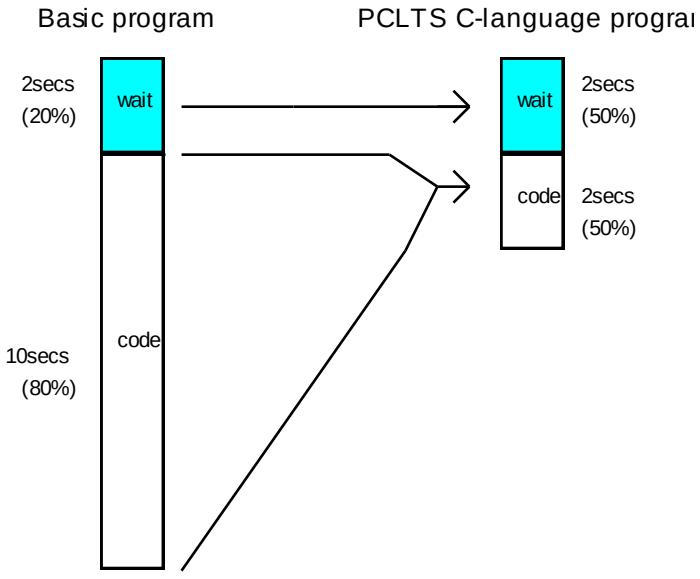
Functions in C provide code modularity and also allow using variables with a scope limited to that particular function. These variables are local and invisible from other functions. In LTS Basic this is not allowed since all variables are global in scope.

Reducing Test Times

Wait Time as a Percentage of Total Test Time

By converting test programs to the PCLTS system, test times will be reduced considerably (anywhere from 1/3rd of the original, which is typical for op-amps, to 1/10th, achieved on some multiple DACs). Test times of test programs can be split into actual instruction execution time and time spent in **wait** statements. The PCLTS system reduces the instruction execution time, but, of course, the wait time remains constant. So programs that had original LTS Basic test times consisting of 20% wait time and 80% code execution time, after conversion may consist of 50% wait time and 50% code execution time.

With the test program in the LTS Basic format, optimizing the wait time did not have a dramatic effect (percent wise) in reducing the test time. After conversion, the 50% wait time is a very significant proportion of the total time and optimizing it reduces the final test time even further by a significant percentage.



"Wait" time as a percentage of the test program.

Use *fast_meas()* and *fast_diff()*

One way of reducing test times is to use the *fast_meas()* and *fast_diff()* system functions instead of their regular counterparts. As opposed to their LTS Basic equivalents, these quick functions use all the measurement system calibration factors and do set the system alarm if necessary, so they are very safe to use. They should be used together with the *set_gain_relays()* function. The "fast" functions take one tenth of the time that the regular measure functions do to execute.

Please refer to the **PCLTS system library functions** chapter for more details on usage of these functions.

The *msd()* function is no longer needed

On the standard LTS-2020, the CPU board is active at all times performing operating system housekeeping tasks and therefore induces noise on the analog (measurement) boards. To reduce the effect of this noise, the digital bus is disconnected ("tri-stated") from the measurement board during the critical A-to-D conversion of a measurement. The MSD (Measurement Settling Delay) statement instructs the system to wait a certain amount of time between disconnecting the digital bus and strobing the A-to-D converter to allow the digital lines to fully settle.

With the PCLTS system, however, there are no active digital boards while a conversion is taking place since the operating system is running on the PC and the LTS backplane digital lines are used only for test-related acticity (not operating system housekeeping).

Still, for consistency reasons, the msd() function has been implemented in the PCLTS system, and performs the same actions. But using it in a test program will increase the test time significantly, so it is strongly recommended not to use it.

If an LTS Basic program that did use the MSD statement is being converted to PCLTS, it may be necessary to add *lts_wait()* calls before some measurements since the MSD may have been needed for other voltages to settle (not for digital bus settling).

The Source Profiler

The PROFILER.EXE program is a utility provided with Microsoft's C compiler package that has the capability of producing a report detailing the time spent on each section of the test program during execution. This is a very useful tool for optimizing test time.

Please refer to Microsoft's documentation for details on usage of this utility.

Using CONVERT

The PCLTS system provides a fairly quick way to convert existing LTS Basic test program libraries into its PCLTS C-language equivalents. This feature avoids re-designing test hardware and test methods for products that have long been in production while considerably reducing their test times and providing new system features.

Transferring the LTS Basic File to the PC

The first step in converting an existing LTS Basic program into a PCLTS C-language equivalent, is to transfer the file to the PC's hard disk. This can be done by using the PC as a terminal connected to either port 1 or port 2 of a non-converted LTS console.

After loading ADOS in the normal fashion, the test program to be converted should be NLOADed on the LTS . The PC should have a communications program such as XTALK VI, Smartcomm, or the terminal emulator included in Microsoft Windows, loaded to transform it into a virtual terminal. A pin-to-pin serial cable is then used to connect the PC to the LTS console through their respective serial ports (port 1 on the LTS). The appropriate communications parameters must be set on the terminal emulator software. These will depend on the parameters that are set on the copy of ADOS that has been loaded on the LTS. The PC then acts as the LTS's Qume terminal and is in control of the console.

At this point the communications program must be set in "capture to file" mode, and the user must type LIS from the PC's keyboard. This LTS command will list the entire program, which will be captured to the hard disk by the terminal emulator program.

There are other variations on this method (using port 2, for example, and leaving the Qume terminal connected to port 1 while listing the program to UNIT3).

Preparing the LTS Basic Program for Conversion

Once the LTS Basic test program file has been transferred to the PC, there is some minor editing that should be done to it before processing it with the CONVERT program. Since LTS Basic files are ASCII, any text editor may be used to edit them. Examples of text editors are EDIT (provided with MS DOS 5.0 and higher) and PWB (provided with Microsoft's C compiler). If word processors are used, they must be used in *text* mode, or else they will add special formatting characters to the file which will result in errors.

Housekeeping Code

LTS Basic files normally contain large sections of code related to data logging, binning, summary sheets, and related topics ("housekeeping" code). Before converting a program from Basic to PCLTS C-language, it must be stripped of all the housekeeping code that does not directly relate to the testing itself. An example of this is the entire "function switch" portion (normally found at lines 30000 through 32767). In the PCLTS system, all these type of functions are performed by the system software and not by the test program.

Test Parameters Statements

Test parameter statements ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) should be left as part of the program, since the conversion program will use this data to generate the converted test program and its associated limit table (if the user so requests it).

Test Numbers

The PCLTS system automatically increments the system variable **tnum** after each test, so there is no need for the user to do this in the test program. However, the **tnum** system variable may be set to a number at any point in the test program. An example of this is skipping two test numbers by setting tnum to 20 after test number 17.

Regarding Subroutines

All LTS BASIC subroutines should end in a final RETURN statement on the last line of the subroutine's body. This RETURN must be the first instruction in that last line (it doesn't need to be the only instruction in that line). The CONVERT program identifies the start of a subroutine's body by the GOSUB destination line number and identifies the end by the first RETURN found at the beginning of a line. For this reason, the subroutine code may not jump forward past the final RETURN. Other alternate RETURNs inside the subroutine are allowed as long as they are not the first statement in that line**.**

An example of valid code could be:

```
100 VSA=3:: DDB=120H
200 GOSUB 500:: B=3
300 MEAS
500 DCB=3:: B=C+2              ! start of subroutine body
600 DDB=A+300H
700 IF P=2 THEN RETURN     ! alternate RETURN inside subroutine
800 VSD=2.5::A=5
900 RETURN:: A=C+4   !  this RETURN marks the end of the routine
950 VSA=6
```

An example of invalid code would be:

```
100 VSA=3:: DDB=120H
200 GOSUB 500:: B=3
300 MEAS
350 GOSUB 600    ! The subroutine is entered at a second line number
500 DCB=3:: B=C+2      ! start of subroutine body
600 DDB=A+300H         ! another entry point to the subroutine (ERROR)
650 IF C=2 THEN GOTO 980 ! jump outside subroutine body (ERROR)
700 IF P=2 THEN RETURN     ! alternate RETURN inside subroutine
800 VSD=2.5::A=5
900 B=4:: RETURN ! ending RETURN not the 1st instruction (ERROR)
950 VSA=6
980 DCB=5
```

In the above example, the RETURN at line 900 must be the first statement in that line, or else the CONVERT program will not recognize it as the end of the subroutine body.

In general, if the LTS Basic test program is reasonably well structured in it's use of subroutines, there shouldn't be any problems. On the other hand, if it JUMPs from one line to another without respecting subroutine beginnings and endings, then you will need to correct the offending code before processing the file with the CONVERT program.

Maximum Line Width

The maximum line width of LTS Basic source files supported by the CONVERT program is 120 chars (1.5 lines). In any case, the LTS-2020 interpreter does not allow more than this length, so it is unlikely that users would have existing Basic programs with more than 120 characters per line.

Nested IF Statements

Only two levels of nested IFs are supported. This implies having two IF statements on a single line in the LTS Basic program. It is very unlikely that the user would have more than two IFs on a single line.

Line Numbers

Every line of text in the LTS Basic source file must begin with a line number (or be an empty line). Again, given that ADOS does not allow otherwise, it is very unlikely that users would have test programs with lines that do not begin with a line number.

Running CONVERT.EXE

The CONVERT program will take an LTS Basic test program file and create a PCLTS C-language test program. The Basic file must first undergo some minor

editing before processing it using CONVERT (see **Preparing the LTS Basic Program for Conversion** section above).

CONVERT.EXE is run from the DOS prompt, and it takes the names of the source Basic file and target C file from the command-line. It also has three additional optional parameters that can provided from the command-line. The syntax is as follows:

CONVERT <*source filename*> <*target filename*> [/l] [/emb] [/nopar]

The "<>" brackets mean that the parameter is required; the "[]" brackets mean that the parameter is optional. The brackets themselves should not be included when actually typing the command at the DOS prompt.

The command-line parameters are as follows:

source filename This is the name of the file containing the LTS Basic program to be converted. The file name may contain path information if needed. It should be an ASCII file.

target filename This is the name of the file where the CONVERT program will write the resulting C-language program and should therefore have a ".c" extension. The file name may contain path information if needed. It will be an ASCII file.

/l This is an optional switch which instructs the CONVERT program to display on the screen the Basic line number of the line currently being processed. It is helpful in troubleshooting problems in case the CONVERT program aborts execution without successfully finishing the conversion.

/emb This is an optional switch which instructs the CONVERT program to embed the test name, units, limits, formatting decimals and fail bin as part of the test program using the *test_parameters* function. If this switch is not used, CONVERT will put this information in an external limit table file (using the *auto_test_parameters_and_bin* function) instead of embedding it in the program. This second method (limit table) is more convenient and elegant.

In both cases described above, the statements related to test parameters ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) will be removed from the test program.

/nopar This is an optional switch which instructs the CONVERT program to use neither embedded test information nor a limit table. In this case, the statements related to test parameters ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) will not be removed from the test program.

This switch has only been included in case the user wants to handle these variables in a way different than the PCLTS system does. Most users will not want to use this switch.

Conversion Examples

The following three conversion examples illustrate the use of CONVERT. A very simple LTS Basic program is converted using different command line switches.

LTS Basic program:

```
  300 REM ********* Filename = "DAC01.BAS" ********************
  500 REM *************** TEST #1 ****************
 1000 BASE 01860H:: CRF[8]=08H:: GON SA:: VSA=15
 1100 PCLASS 1 TO 2:: $TAG[0]="Output Voltage":: FCLASS 1 TO 4
 1200 $FMT[0]=" SS.99":: LL=4.9:: UL=5.1:: $UNT="V"
 1300 SELECT LB,LD:: MAX=5:: MEAS:: WAIT 0.001
 1400 VL=RES:: GOSUB 32000
 1500 REM *************** TEST #2 ****************
 2000 BASE 01840H:: CRF[0]=08551H:: GON SR:: VSR=5
 2100 $TAG[0]="Input Resistance":: FCLASS 1 TO 8
 2200 $FMT[0]=" SS.999":: LL=2.45:: UL=2.65:: $UNT="Kohms"
 2300 SELECT LA,LC:: MAX=5:: MEAS 10:: WAIT 0.001
 2400 VL=RES/1000:: GOSUB 32000
```

Example 1.- *Using a limit table*

CONVERT defaults to using a limit table if the ***/emb*** switch is not used. All test parameter statements ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) will be extracted from the program and replaced by a call to the *auto_test_parameters_and_bin* function wherever the original program had a GOSUB 32000 statement. For this example, the command-line arguments for executing CONVERT would be:

```
CONVERT DAC01.BAS DAC01.C
```

Where DAC01.C is the name of the C-language converted program. The resulting DAC01.C file would be:

```
void test_program()
{
     /*  *************** TEST #1 **************** */
     base(0x01860); crf(8,0x08); gon(SA); vsa(15);
     pclass(1,2);
     select(LB,LD); maxx(5); meas(1,NOSYNC); lts_wait(1);
     vl=res;
     auto_test_parameters_and_bin(vl);
     if(check_fail) return;
     /*  *************** TEST #2 **************** */
     base(0x01840); crf(0,0x08551); gon(SR); vsr(5);
     select(LA,LC); maxx(5); meas(10,NOSYNC); lts_wait(1);
     vl=res/1000;
     auto_test_parameters_and_bin(vl);
     if(check_fail) return;
}  /* end of 'test_program()' function */
```

A corresponding limit file will automatically be created as follows:

Limit Table

Product : XXX-1234
Rev : 01-01-94
Engineer: John Doe

Pass Bin	Test# Beg	End	Test Name	Units	Fmt Decs	Fail Bin	QA MIN	QA MAX
1	??	??	Output Voltage	V	2	5	4.9	5.1
1	??	??	Input Resistance	Kohms	3	9	2.45	2.65

The corresponding test numbers must then be entered by editing the limit file (using a text editor). Since no test sequences with common parameters are used in this example, the beginning and ending test numbers on each of the two tests would be the same (1 1 for the first row and 2 2 for the second).

Example 2.- *Using embedded test parameters*

If the ***/emb*** switch is used, CONVERT will not create a limit table. All test parameter statements ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) will be extracted from the program and replaced by a call to the *test_parameters* and *bin_current_test* functions wherever the original program had a GOSUB 32000 statement. For this example, the command-line arguments for executing CONVERT would be:

CONVERT DAC01.BAS DAC01.C **/EMB**

Where DAC01.C is the name of the C-language converted program. The resulting DAC01.C file would be:

```
void test_program()
{
     /*  ***************** TEST #1 ***************** */
     base(0x01860); crf(8,0x08); gon(SA); vsa(15);
     pclass(1,2);
     select(LB,LD); maxx(5); meas(1,NOSYNC); lts_wait(1);
     vl=res;
     test_parameters(vl,"Output Voltage","V",4.9,5.1,"2",5);
     bin_current_test(vl);
     if(check_fail) return;
     /*  ***************** TEST #2 ***************** */
     base(0x01840); crf(0,0x08551); gon(SR); vsr(5);
     select(LA,LC); maxx(5); meas(10,NOSYNC); lts_wait(1);
     vl=res/1000;
     test_parameters(vl,"Input Resistance","Kohms",2.45,2.65,"3",9);
     bin_current_test(vl);
     if(check_fail) return;
}  /* end of 'test_program()' function */
```

Example 3.- *Using neither embedded test parameters nor limit table*

If the ***/nopar*** switch is used, CONVERT will not extract the test parameter statements ($TAG=, $UNT=, $FMT=, LL=, UL=, and FCLASS=) from the test program. Also, all GOSUB 32000 statements will be left untouched. For this example, the command-line arguments for executing CONVERT would be:

CONVERT DAC01.BAS DAC01.C **/NOPAR**

Where DAC01.C is the name of the C-language converted program. The resulting DAC01.C file would be:

```
void test_program()
{
     /*  ***************** TEST #1 ***************** */
     base(0x01860); crf(8,0x08); gon(SA); vsa(15);
     pclass(1,2); tag[0]="Output Voltage"; fclass(1,4);
     fmt[0]=" SS.99"; ll=4.9; ul=5.1; unt="V";
     select(LB,LD); maxx(5); meas(1,NOSYNC); lts_wait(1);
     vl=res; line_32000_subroutine();
     /*  ***************** TEST #2 ***************** */
     base(0x01840); crf(0,0x08551); gon(SR); vsr(5);
     tag[0]="Input Resistance"; fclass(1,8);
     fmt[0]=" SS.999"; ll=2.45; ul=2.65; unt="Kohms";
     select(LA,LC); maxx(5); meas(10,NOSYNC); lts_wait(1);
     vl=res/1000; line_32000_subroutine();
}  /* end of 'test_program()' function */
```

The CONVERT.RPT File

After every successful execution of the CONVERT program, the system creates a file called CONVERT.RPT. This file is an ASCII file and it contains a report of the conversion results consisting mainly of statistics and label cross-references. It can be helpful in debugging conversion anomalies. It can be printed from the DOS command line by typing

PRINT CONVERT.RPT

This file is overwritten every time that a the CONVERT program is run, so it is a good idea to print it or copy it to another file if the information in it will be needed at a later time.

The following is an example of a CONVERT.RPT file:

```
            =========[ File Conversion Report ]========
                                                  Date: 04/11/93
                                                  Time: 21:54:19

  Source file name          : OP_07.BAS
  Target file name          : OP_07.C

  Number of subroutines     : 9
  Number of GOTO labels     : 8
  Number of converted lines : 484
```

```
The following subroutine bodies were not found :

            Subroutine at line number 8640

The following GOTO destinations were not found in the source file:

            Line number 5290    (label L9)


Labels cross-reference:

           Label   Line number
           -----   -----------
            L1          1650
            L2          1700
            L3          1850
            L4          2000
            L5          3000
            L6          3250
            L7          4500
            L8          5000
```

Editing the Converted PCLTS C-Language File

Once the C-language file has been produced by the CONVERT program, the user must do some final editing to ensure a successful compilation of the test program.

Unused Variables and Statements

There probably will be several variables and statements that where used for "housekeeping" tasks in the LTS Basic programs that will no longer be needed as part of the test program. In the PCLTS system all the data logging, binning, summary information etc., is handled by the system and not by the test program.

A few examples of these variables are:

UNIT
QCK
STAT
BIT
BUSDAT
CALFLG

Some of these (like STAT, for example) have been replaced by other PCLTS system function calls. Please refer to the "LTS Basic Cross Reference Table" in appendix A for other variables and statements.

Deleting Test Number Increments

The PCLTS system increments test numbers (**tnum** system variable) automatically after every test, so there is no need to do this in the test program. Furthermore, if it is done in the test program, **tnum** will end up being incremented twice.

There may be situations, however, where the user may want to skip a few test numbers. This may be done by setting **tnum** to a higher value at any point in the code. For this reason, the CONVERT program does not delete all occurrences of

TNUM, but rather leaves it to the user to decide which TNUM statements, if any, should be kept. This editing can be done prior to or after the conversion. The CONVERT program does not use the value of TNUM at any point.

The system also initializes **tnum** to a value of 1 before executing the test program.

Note that only *integer* test numbers are supported.

Variable Type Declarations

An important part of the post conversion editing is the variable declarations. LTS Basic does not have true type declarations. All variables are stored in the same number of bytes (six bytes). In C-language, programs must explicitly declare all variables used in the program.

It is up to the user then to scan the test program and decide what type to use for each one of the variables used. Note that variables like **res**, **tnum**, **snum**, **savo**, etc, are system variables already defined and must not be defined again in each test program. For a complete list of system variables please refer to the **PCLTS System Variables** chapter.

Most variables will be declared as either type **int** or **double**, for integer and double-precision floating point, respectively. The user should read the C compiler manual or any C-language textbook to become familiar with the different data types available.

A final warning regarding data types:

Consider the following example:

```
int num_a;      /* integer type declaration */
double num_b;  /* floating point type declaration */

num_b=10;
num_a=num_b/3.0;
```

After execution, *num_a* will have a value of 3, and not 3.333333, since it has been declared as an integer. The C compiler will not consider this assignment an error, so it may easily go undetected, with potential disastrous results on test yields!

An Important Note on Array Subscripts

Arrays in C-language are defined differently than in Basic. In Basic, the number that appears in the DIM declaration represents the last index of the array, while in C-language, the number in the type declaration represents the size of the array. Both Basic and C start their array indexes at "0".

As an example, consider an array of integers named ABC and with a maximum size of five elements:

In Basic:	In C-language:
DIM ABC[4] !size of array is 5	**int abc[5]**; /* size of array is 5 */

Both arrays have the same range of indexes: abc[0] through abc[4], but they are declared differently.

The user must be very careful when declaring array sizes during a Basic-to-C conversion. All the array references inside the test program body may be left unchanged, but the declarations in C-language must be one number larger than their DIM Basic counterparts. Note that if the declaration is left the same, the C compiler will not reject it, but during run-time the array will overwrite memory space that has not been assigned to it by the system, often with catastrophic results. These type of problems are very hard to debug.

Replacing *goto*s

LTS Basic is not known for lending itself well for structured programming. It lacks many constructs that C and other languages have. It is very easy to abuse the use of **goto** statements and labels. The CONVERT program leaves these **goto**s untouched, since it is fairly complex to automate the process of replacing them with more elegant structures.

It is a good idea to spend some time cleaning up the programs by replacing **goto**s for **while**, **do-while**, **switch-case**, and nested **if** statements. In theory, there is no need for **goto**s and every program can be stripped of them by using other available instructions.

Other Utilities

Running *Verify Setup*

The PCLTS software has a built-in system for verifying the test hardware integrity before testing a parts lot. This function is exercised by using the ***Verify Setup*** option of the LTSSHEL menu. This is how it works:

The first step is to prepare a *reference file* for a particular unit of the device being tested. A reference file is simply a file with representative averages for a particular device of the results of all the tests in the test program. Reference files are created using the CORRLGEN utility (described below). The reference file created for a device will represent what the test results for that device should be.

The operator must then insert the reference device in the hand socket or handler, and select the ***Verify Setup*** option. The system will test the device and datalog the results to the screen and to a temporary datalog file (named CORRLTMP.STD and later deleted). The results for each test in the temporary file is then compared with the reference file for that device, and if they do not correlate appropriately, the setup verification will fail. The criteria for correlation is read from a third file called, *correlation criteria* file. This file is created by the user and normally contains the system repeatability, accuracy, and system-to-system repeatability for each one of the tests in the test program.

The system generates a report showing the results of each test, the reference value for each test, the actual difference between the two values and the correlation criteria, for every test that failed or was within 20% of failing. The latter is shown to note the tests that may be potential correlation failures.

Correlation Criteria File Format

Correlation criteria files must have a file name formed by the same prefix as the test program, but with the ".COR" extension. They must reside in the directory designated in the LTSSHEL.INI file (the settings in this file are modified by using the ***System Configuration*** option of the Development Menu in the LTSSHEL).

The file format is ASCII and very similar in form to the limit table files. They consist of three columns: the test number (beginning and ending), the test name, and the criteria values. The test name is included only for readability when printing a file. Just like limit files, correlation files may have any number of comment lines before the start of the actual table. The concept of beginning and ending test numbers is also the same as for limit tables. The following is an example of a correlation criteria file:

Correlation Criteria

Product: DAC-1234
Engineer: John Doe
Rev Date: 5/3/94

Test # Beg	Test # End	Test Name	Criteria
1	1	Supply Current	0.001
2	2	Reference Voltage	0.005
3	7	Input Current	0.3
8	8	Input Resistance	100
9	9	Offset Voltage	0.004
10	10	Output Current	0.4

A Correlation Criteria File

The number of tests in the correlation criteria file must match the number of tests in the corresponding limit file.

CORRLGEN.EXE Utility

The CORRLGEN program is used to generate reference files for correlation devices. These files are then used before testing a parts lot to verify the test setup. To run this program simply type

CORRLGEN

from the DOS prompt. The program is menu-driven and very simple to use.

Trial Files

Before generating the actual reference file, several trial files must be created. A trial file is simply a datalog file with data corresponding to a single execution of the test program. The average of a set of a trial files for a given device constitutes the reference file for that device. In order for the reference file to be truly representative average of a particular device's data, the trial files should be generated using different combinations of all the hardware components that will be later used with the device.

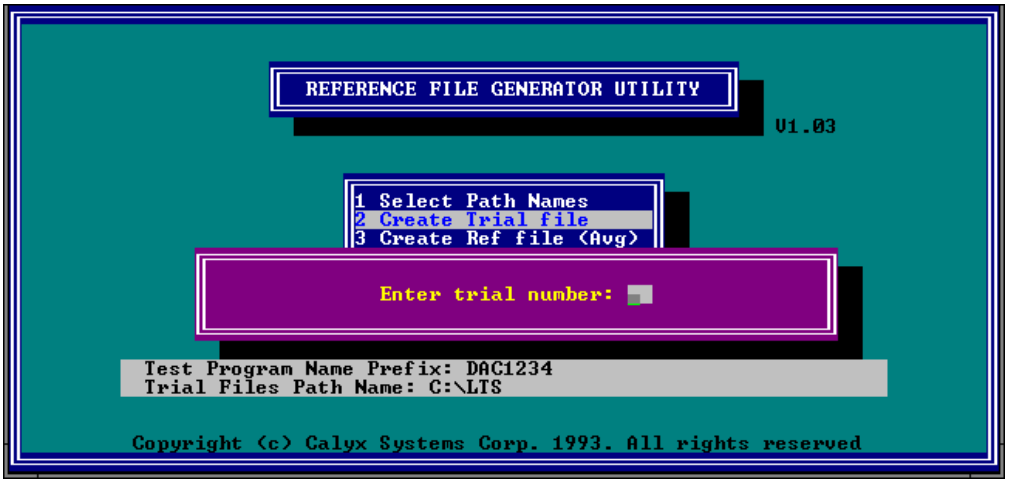
Trial File Format

Trial files are ASCII datalog files, similar to a regular datalog file converted from STDF to ASCII, but without the header. These files can be printed and edited.

The following is an example of a trial file:

```
T#1          48.1 mA     Supply Current @5V
T#2          50.0 mA     Supply Current @6V
T#3         17.32 Kohms  Input Resistance A
T#4         18.00 Kohms  Input Resistance B
T#5         -0.15 LSB    DNL (Major Carries)
T#6         -0.05 LSB    DNL (Major Carries)
T#7          0.05 LSB    DNL (Major Carries)
T#8          0.15 LSB    DNL (Major Carries)
T#9          0.25 LSB    DNL (Major Carries)
T#10         0.35 LSB    DNL (Major Carries)
T#11         0.45 LSB    DNL (Major Carries)
T#12         0.55 LSB    DNL (Major Carries)
T#13       -0.008 %/%    PSRR
Pass Bin 1
```

Trial files are named with the same filename prefix as the test program that generates them, but with the "**.Tn**" extension, where **n** is the trial number and can be any integer from 1 to 99. For example, the file for trial number 8 using the DAC_1234.EXE test program would be called DAC_1234.T**8**.



Reference File Format

Once a set of trial files has been created, the reference file may be generated. The system will prompt for a device number and will then use that number to form the filename of the reference file. This will have the same prefix as the test program, but will have the "**.Dn**" extension, where **n** is the device number. So the reference file for device number 12 created with the DAC_1234.EXE test program will we named DAC_1234.D12.

The reference file is an ASCII file with the average value of the results of the tests in the test program. Each line consists of the test number followed by a comma and the average value for that test. This file may be printed and edited.

The following is as example of a reference file:

```
 1 ,    17.5000000
 2 ,    14.8666666
 3 ,    17.6333333
 4 ,      .0000000
 5 ,     1.9833333
 6 ,      .0000000
 7 ,      .0000000
 8 ,   185.6666666
 9 ,     1.1630000
10 ,     8.0000000
11 ,    11.0000000
12 ,    16.0000000
13 ,    31.0000000
14 ,    20.0000000
15 ,    22.0000000
16 ,    25.0000000
17 ,    27.0000000
18 ,    30.0000000
19 ,    31.0000000
20 ,    30.0000000
21 ,    30.6666666
22 ,    29.3333333
```

PRINTCAL.EXE Utility

The PRINTCAL utility is provided to print the LTS calibration factors that resulted from the last hourly system calibration. It is analogous to the CAL FACTORS utility included in the LTS-2020 ADOS system disk. To execute, simply type

PRINTCAL | MORE

from the DOS command line (the "| MORE" part of the command will instruct DOS to "pipe" the output to the DOS MORE utility and as a result, the output will be displayed one page at a time). The calibration factors together with their corresponding low and high limits will be printed to the screen.

If a hard copy is desired, type

PRINTCAL > PRN

This will redirect the output to the printer instead of the screen. The output can also be sent to a file by typing:

PRINTCAL > FILENAME

The following is a typical output from the PRINTCAL utility:

```
        +-------------------------------------------------+
        |                                                 |
        |              LTS CAL FACTORS  V1.0              |
        |                                                 |
        +-------------------------------------------------+
         Last hourly calibration: Sat Mar 17 19:22:39 1993

          Low Limit     Actual Value   High Limit   Parameter
          ---------     ------------   ----------   ---------
0      2043.000000    2049.000000   2053.000000     ADOFS
1         0.004858       0.004886      0.004910     ADLSB
2        -0.001000       0.000091      0.001000     GBOFS
3        -0.000500       0.000012      0.000500     FOLOFS
4        -0.000500       0.000027      0.000500     IAOFS
5        -0.000200       0.000001      0.000200     IACMG
6         0.998000       0.999531      1.002001     IAGAIN
7        -0.005200       0.000434      0.005200     RD10UO
8         0.002436       0.002442      0.002446     RDLSB
9        -0.000250       0.000053      0.000250     RDSPOS
10        4.988779       5.000895      5.011221     RDBT00
11        2.493779       2.500310      2.506221     RDBT01
12        1.246280       1.250287      1.253720     RDBT02
13        0.622530       0.625147      0.627470     RDBT03
14        0.310655       0.312778      0.314345     RDBT04
15        0.154717       0.156405      0.157783     RDBT05
16        0.076749       0.078217      0.079501     RDBT06
17        0.037765       0.039149      0.040361     RDBT07
18        0.017741       0.019592      0.020259     RDBT08
19        0.008520       0.009773      0.011000     RDBT09
20        0.003653       0.004881      0.006113     RDBT10
21        0.001221       0.002471      0.003653     RDBT11
22       -5.020101      -4.999712     -4.979899     RD10BF
23       -0.010000       0.000851      0.010000     RD20UO
24        1.992499       1.999558      2.007501     RD21G
25      -10.040200      -9.996123     -9.959800     RD20BO
26        0.000562       0.000623      0.000695     ND2LSB
27        0.097000       0.103641      0.113000     ND1LSB
28       -0.088000      -0.000913      0.088000     SAOFS
29       -0.088000       0.002807      0.088000     SBOFS
30       -0.094000      -0.002534      0.094000     SCOFS
31       -0.094000      -0.004861      0.094000     SDOFS
32       -0.091800      -0.083664     -0.075700     VSDLSB
33       -0.091800      -0.083255     -0.075700     VSCLSB
34        0.077000       0.082457      0.090000     VSALSB
35        0.077000       0.086214      0.090000     VSBLSB
36      -10.970000     -10.625477    -10.419999     SROFS
37       10.000000      10.629297     12.415001     SRBT00
38        4.900000       5.314348      6.180000     SRBT01
39        2.429999       2.657097      3.120001     SRBT02
40        1.209999       1.328455      1.580001     SRBT03
41        0.579000       0.664320      0.815000     SRBT04
42        0.270000       0.332136      0.431000     SRBT05
43        0.115000       0.165991      0.240000     SRBT06
44        0.038000       0.083038      0.144000     SRBT07
45        0.059000       0.064236      0.071000     SRBT08
46        0.029300       0.032013      0.035500     SRBT09
47        0.014400       0.016099      0.018000     SRBT10
48        0.006970       0.008003      0.009300     SRBT11
49        0.003250       0.003999      0.004930     SRBT12
50        0.001390       0.001909      0.002740     SRBT13
51        0.000460       0.001036      0.001640     SRBT14
52        0.000000       0.000511      0.001100     SRBT15
53        0.039800       0.041017      0.044000     VTHLSB

Error Code=0H
```

The Cal Factors

The following is a description of the fifty-four cal factors:

ADOFS	Offset voltage of measurement system ADC
ADLSB	LSB size of measurement system ADC
GBOFS	Offset of measurement system's gain buffer
FOLOFS	Offset of measurement system's voltage follower
IAOFS	Offset voltage of the instrumentation amplifier found in the system measure board.
IACMG	Common-mode gain of the instrumentation amplifier found in the system measure board.
IAGAIN	Gain of the instrumentation amplifier found in the system measure board.
RD10UO	Ref DAC offset when in 10-Volt span, unipolar mode
RDLSB	Reference DAC LSB size
RDSPOS	Reference DAC superposition error
RDBT00...RDBT11	Contribution in Volts of each one of the twelve bits of the Ref DAC.
RD10BF	Ref DAC offset when in 10-Volt span, bipolar mode
RD20UO	Ref DAC offset when in 20-Volt span, unipolar mode
RD21G	Ref DAC 20V/10V gain
RD20BO	Ref DAC offset when in 20-Volt span, bipolar mode
ND2LSB	Null DAC 2 (Fine) LSB size
ND1LSB	Null DAC 1 (Coarse) LSB size
SAOFS	Offset voltage of source A
SBOFS	Offset voltage of source B
SCOFS	Offset voltage of source C
SDOFS	Offset voltage of source D
VSDLSB	LSB size of source D

VSCLSB	LSB size of source C
VSALSB	LSB size of source A
VSBLSB	LSB size of source B
SROFS	Offset voltage of source R
SRBT00...SRBT15	Contribution in Volts of each one of the sixteen bits of the source R DAC.
VTHLSB	LSB size of VTH voltage source found in the Digital I/O system board.

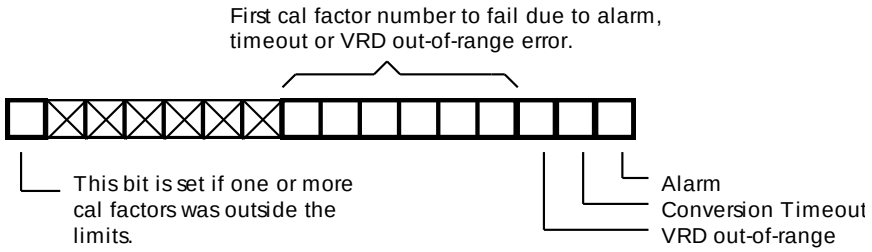
When Calibration Fails

If for any given calibration factor, the value measured falls outside of the allowable limits, an "F" will be printed to the right of the failing parameter(s) on the **Actual Value** column. The failing parameter may help the user determine which system board and what part of the board is defective. In addition to looking at failing parameters, the error code (described below) can be examined to gain further understanding of the failure mode.

The Error Code Word

When PRINTCAL is run, an error code is printed at the end of the listing. Normally, this number should be 0, indicating a successful calibration.

If calibration fails, the error code will result in a non-zero value. The following is a description each one of the bits of the 16-bit value:



Error Code Word Format

Bit 0 (LSB) will be set if the measurement system alarm was set at any stage during calibration.

Bit 1 will be set if the measurement system's ADC timed-out during a conversion.

Bit 2 will be set if the Reference DAC is set to a value outside of the allowed range at any point during calibration.

Bits 3 through 8 will contain the cal factor number of the first parameter to fail due to a system alarm being set, the ADC timing-out, or the system trying to set the Ref DAC to an illegal value. **These bits will not be set if calibration failed due to a parameter being outside the limits**.

Bits 9 through 14 are reserved for future use (unused at this time).

Bit 15 will be set if one or more cal factors was outside the allowable limits.

Examples:

Error code = 8000H

This means that one or more cal factors were outside the limits. The listing of the cal factors would show which limits failed (the ones with an "F" by them).

Error code = 8031H

This means that one or more cal factors were outside the limits, and furthermore, cal factor number 12 (000110), which corresponds to RDBT02, set the measurement system alarm.

Error code = 34H

This means that cal factor number 12 (000110) tried to set the Ref DAC to an out-of-range value. Even though this happened, all the cal factors resulted in values that were within the allowable limits (bit 15 was not set). However, calibration still fails.

BRDINSTL.EXE Utility

This is a very simple but handy utility. When executed, it prints a message on the DOS screen indicating if the CPU Emulator board is installed in the PC. To execute simply type

```
BRDINSTL
```

from the DOS command line.

HDWCLR.EXE Utility

This executable program is provided to clear the LTS hardware settings from the DOS command line. To execute simply type:

```
HDWCLR
```

from the DOS prompt. This is useful in case the user exits the debugger (Codeview) without terminating execution of the test program (while in a trap for example). Note that an alternative to this would be to execute the *hdwclr()* function from the Codeview command window before exiting.

PCLTS System Library Functions

The entire set of LTS Basic functions and statements has been ported to C-language functions and variables. Some of the functions (like DFORM and PSAVE, for example) are not applicable to the PCLTS system. This type of functions are performed by the DOS operating system, outside the PCLTS shell. Please refer to the LTS Basic-PCLTS cross reference table to find the proper equivalent C function or variable.

Functions Listing

The following is a comprehensive listing of the PCLTS system functions. For each function, it includes a function prototype, a description, and also outlines the differences between the original LTS Basic function and the PCLTS version of it. A usage example is also included for further clarification.

aclass

Prototype:

void **aclass**(int a, int b)

Description:

Specifies which classes should be disqualified when the alarm flag is set. It should be used together with pclass() and fclass() at the beginning of the test program to set up the classification system. The classes to disqualify are classes **a** through **b**, both included. Allowed class values are 1 through 32.

Differences with LTS Basic:

This function is equivalent to "ACLASS a **TO** b " in Basic, but not **"**ACLASS a**,**b". The latter form of the function is not supported. Also, multiple sets of bins may not be listed with a single call of aclass(). So the equivalent to "ACLASS 3,5 TO 7,9" in Basic would be the following in PCLTS C-Language:

```
aclass(3,3); aclass(5,7); aclass(9,9);
```

auto_test_parameters_and_bin

Prototype:

int **auto_test_parameters_and_bin**(double a)

Description:

Sets the necessary parameters for the current test and also tests the value **a** against the test limits. The individual parameters used (pass bin, test name, units, formatting

decimals, fail bin, and limits) are the limit table values previously stored in memory. This function should only be called when a valid limit table exists, or else the call will result in a run time error.

Calling this function is equivalent to using the

test_parameters();
bin_current_test();

sequence for the case when no limit table is used.

Example:

```
meas(1,NOSYNC);                      /* measure */
auto_test_parameters_and_bin(res);   /* set parameters and bin current test */
```

base

Prototype:

void **base**(unsigned short a)

Description:

Sets the CRU base address to the value **a** for all subsequent *crf()* and *crb()* calls. It also sets the system variable *base_value*, which can be checked by the user any time during the program.

Differences with LTS Basic:

The PCLTS system sets the *base_value* system variable which can be accessed by the user to check the last base value set (this is especially useful during a debugging session).

Example:

```
base(0x1800);      /* set value of CRU base to 1800 Hex */
```

bin_current_test

Prototype:

int **bin_current_test**(double a)

Description:

This function performs the following tasks:

1.- Sets the system variable fail if the value a is outside the limits set in the last test_parameters() call.
2.- Calls the test() function.
3.- Clears the system variable test_downgraded.
4.- Increments the system variable tnum.

This function should be called immediately after test_parameters() at the end of every test in the test program.

Example:

```
meas(3,NOSYNC);
test_parameters(res,"Idd", "mA", 0.01,25, "3",9);
bin_current_test(res);
```

check_fail

Prototype:

int **check_fail**(void)

Description:

Allows test program to terminate if device has failed. It returns TRUE if the system variable *device_failed* is set and if the "force through failures" option is not set. It must be the called at the end of every test on the test program and must execute a *return* if the function returns a TRUE value.

Example:

```
test_parameters(res,"Idd", "mA", 0.01,25, "3",9);
bin_current_test(res);
check_fail() return;                /* terminate the test prog if device failed */
```

clas

Prototype:

int **clas**(void)

Description:

Returns the lowest non-zero class which has not yet been disqualified. If all classes have been disqualified, then it returns zero.

Differences with LTS Basic:

None, except for the difference in spelling. (*class* is a reserved word in Microsoft C++)

cmv

Prototype:

void **cmv**(double a)

Description:

Specifies the expected common mode voltage present at the inputs of the measurement board's instrumentation amplifier. Using this function improves system accuracy when a relatively high common mode voltage appears at these inputs. A correction factor, transparent to the user, is used in the system calculation of the variable *res* returned by the measurement functions.

Differences with LTS Basic:

This function is identical to the Basic CMV statement. The variable *cmv_value* replaces the Basic CMV function.

crb

Prototype:

void **crb**(short a, short b)

Description:

Sets or clears a particular CRU bit. The **a** value is an offset from the base set by the last *base()* function call. It must be between 0 and 127. The **b** value is what that bit is being set to: either 0 or 1.

Differences with LTS Basic:

This function is identical to the Basic CRB statement, with the exception that the offset value may not be a negative value. The *crb_read()* function replaces the Basic CRB function.

Example:

```
base(0x1800);
crb(3)=1;                /* set bit 1806 Hex */
```

crb_read

Prototype:

int **crb_read**(short a)

Description:

Returns the value found at CRU bit **a** relative to the base address set by the last *base()* function call. It returns a value of 1 if the bit is set and 0 if the bit is not set. Note that for this function to return a meaningful value, there must be a readable hardware register at the specified address, or else it will generally return 1 (depending on noise).
This function replaces the LTS Basic CRB function (not statement).

Example:

```
base(0x818);
if(crb_read(3)){  /* if the third data bit of the meas. system A/D is set */
  ddb(3);
}
```

crf

Prototype:

void **crf**(short a, unsigned short b)

Description:

Sets a field of CRU bits to the requested value, starting at the CRU location set by the last *base()* function call. The **a** value is the size of the field in bits. It can be any value between 0 and 15. The **b** value is the value to which the designated bits are to be set.

Differences with LTS Basic:

This function is identical to the Basic CRF statement. The *crf_read()* function replaces the Basic CRF function.

Example:

```
base(0x1820);
crf(0,0x3f11);          /* set the 16 bits starting at 1820 Hex to 3f11 Hex */
```

crf_read

Prototype:

int **crf_read**(short a)

Description:

Returns the value found at CRU field of bits relative to the base address set by the last *base()* function call. The a parameter specifies the size of the field. Note that for this function to return a meaningful value, there must be a readable hardware register at the specified address, or else it will generally return 255 (depending on noise).

This function replaces the LTS Basic CRF function (not statement).

Example:

```
base(0x818);
code=crf_read(8)        /* if the 8-bit field starting at 818 hex (meas. system
                           A/D) */
```

dcb

Prototype:

void **dcb**(unsigned short a)

Description:

Sets the digital control bits to the specified value **a**.

Differences with LTS Basic:

This function is identical to the Basic DCB statement. The variable *dcb_value* replaces the Basic DCB function.

Example:

```
dcb(dcb_value+0x100); /* set the dcb bits to the current DCB value + 100 Hex */
```

dcr

Prototype:

unsigned short **dcr**(void)

Description:

Returns the value of the eight digital control readback bits. This value will be in the range 0 to 255.

Differences with LTS Basic:

None

Example:

```
data=dcr();
```

ddb

Prototype:

void **ddb**(unsigned short a)

Description:

Sets the digital device bits to a specified value **a**.

Differences with LTS Basic:

This function is identical to the Basic DDB statement. The variable *ddb_value* replaces the Basic DDB function.

Example:

```
ddb((i-1)*2);      /* set the DDB lines to the value of (i-1)*2  */
ddb(0x100);        /* set DDB lines to 100H */
```

ddr

Prototype:

unsigned short **ddr**(void)

Description:

Returns the value of the sixteen digital device readback bits. This value will be in the 0 to 65535 (64k) range.

Differences with LTS Basic:

There is only a minor difference. The Basic DDR function returns a signed 16 bit value, and therefore its value will be in the -32768 to 32767 range.

Example:

```
data=ddr();
```

dgate

Prototype:

void **dgate**(int a)

Description:

Simultaneously gates all ddb and dcb bits either on or off . The **a** value must be either ON or OFF.

Differences with LTS Basic:

None

Example:

```
dgate(OFF);
```

diff

Prototype:

double **diff**(int a, unsigned short b)

Description:

Performs and returns the result of a difference measurement. The value **a** is the number of readings to be averaged. The value **b** must be either SYNC or NOSYNC, depending on the use, or no use, of synchronization to the power line. The result of the measurement is returned in the system variable *res*. Valid calls to the *select()* and *null()* functions must always precede a *diff()*.

Differences with LTS Basic:

None, except that the PCLTS *diff()* function always requires two parameters (even if taking only one reading or if no line sync. is requested).

Example:

```
null(1,NOSYNC);
diff(5,NOSYNC);                 /* perform a diff measurement, use 5
                                   readings */
```

disqualify_bin

Prototype:

void **disqualify_bin**(int a)

Description:

Disqualifies bin **a** in the device classification system. This is equivalent to clearing the bit corresponding to class **a** on the *clasv* system variable.

This function is only provided for greater flexibility and ease of use. The system will automatically disqualify the proper bins when either *bin_current_test()* or *auto_test_parameters_and_bin()* are used.

Example:

```
if(res>10.5)   disqualify_bin(1);    /* device can now be at best a bin 2 */
```

dvdt

Prototype:

void **dvdt**(double a)

Description:

Sets the measurement system to take two single measurements **a** seconds apart from each other (minimum is 1 ms). This function only sets the necessary software variables for such a measurement. The actual measurements will not take place until the next *meas()* function call. The result is returned in the *res* system variable.

When called with OFF as a parameter, the measurement system is reset so that the *meas()* following the call is a regular measurement.

Differences with LTS Basic:

None.

Example:

```
select(LA,GD);
maxx(8);
dvdt(0.01);          /* wait 10 ms between the 2 readings */
meas(1,NOSYNC);      /* do the actual readings */
```

Note: If the meas() function called after a dvdt() call has a number of readings greater than 1, the system will ignore the parameter and just do the two single readings.

fast_diff

Prototype:

double **fast_diff**(void)

Description:

A quicker alternative to using the *diff()* function. Performs a difference measurement and returns the resulting value. It differs from the regular *diff()* in the following ways:

1.- The main difference is that *fast_diff()* will not close the measurement system gain relays. It therefore does not need to wait for those relays to settle and is hence faster.

2.- It only allows single measurements (no averaging).

3.- It does not allow power line synchronization.

Since this function does not set the system gain relays, the user must call the *maxd()* and *set_gain_relays()* functions once prior to calling *fast_diff()*.

Differences with LTS Basic:

The main drawbacks of the LTS Basic fast diff call (CALL FD) are that it does not set the measurement system alarm and it does not mathematically compensate for the voltage divider and current sense resistors on the voltage source boards. This is not the case for the PCLTS *fast_diff()* call, which can be safely used.

Another difference is that the LTS Basic programs set the gain relays by calling the "CALL Mx" subroutines. On the PCLTS system, *maxd()* and *set_gain_relays()* are used. Using *maxd()* is easier because you simply specify the maximum expected voltage instead of referring to a cross-reference table to find out what is the maximum voltage corresponding to, for example, CALL M2.

fast_meas

Prototype:

double **fast_meas**(void)

Description:

A quicker alternative to using the *diff()* function. Performs a difference measurement and returns the resulting value. It differs from the regular *diff()* in the following ways:

1.- The main difference is that *fast_diff()* will not close the measurement system gain relays. It therefore does not need to wait for those relays to settle and is hence faster.

2.- It only allows single measurements (no averaging).

3.- It does not allow power line synchronization.

Since this function does not set the system gain relays, the user must call the *maxd()* and *set_gain_relays()* functions once prior to calling *fast_diff()*.

Differences with LTS Basic:

The main drawbacks of the LTS Basic fast meas call (CALL FM) are that it does not set the measurement system alarm and it does not mathematically compensate for the voltage divider and current sense resistors on the voltage source boards. This is not the case for the PCLTS *fast_meas()* call, which can be safely used.

Another difference is that the LTS Basic programs set the gain relays by calling the "CALL Mx" subroutines. On the PCLTS system, *maxx()* and *set_gain_relays()* are used. Using *maxx()* is easier because you simply specify the maximum expected voltage instead of referring to a cross-reference table to find out what is the maximum voltage corresponding to, for example, CALL M2.

fclass

Prototype:

void **fclass**(int a, int b)

Description:

Specifies which classes should be disqualified during the next call to the *test()* function if the *fail* flag (variable) is set. The classes to disqualify are classes **a** through **b**, both included. Allowed class values are 1 through 32.

Differences with LTS Basic:

This function is equivalent to "FCLASS a **TO** b " in Basic, but not **"**FCLASS a**,**b". The latter form of the function is not supported. Also, multiple sets of bins may not be listed with a single call of aclass(). So the equivalent of "FCLASS 3,5 TO 7,9" in Basic would be:

```
fclass(3,3); fclass(5,7); fclass(9,9);
```

gon

Prototype:

void **gon**(unsigned short a)

Description:

Gates on any of the four voltage sources, or reference source. Valid values for parameter **a** are:

SA, SB, SC, SD, SR

Differences with LTS Basic:

The only difference is that *gon()* only accepts one argument. Gating on multiple sources in a single call is not supported. So to duplicate the Basic statement GON SA,SB,SR the proper PCLTS call should be:

```
gon(SA); gon(SB); gon(SR);
```

gof

Prototype:

void **gof**(unsigned short a)

Description:

Gates off any of the four voltage sources, or reference source. Valid values for parameter **a** are:

SA, SB, SC, SD, SR

Differences with LTS Basic:

The only difference is that *gof()* only accepts one argument. Gating off multiple sources in a single call is not supported. So to duplicate the Basic statement GOF SA,SB,SR the proper PCLTS call should be:

```
gof(SA); gof(SB); gof(SR);
```

handlr

Prototype:

void **handlr**(unsigned short a)

Description:

Turns the handler port either on or off. Valid values for **a** are ON and OFF.

Differences with LTS Basic:

None

Example:

```
handlr(OFF);            /* turn handler port off */
```

hdwclr

Prototype:

void **hdwclr**(void)

Description:

Clears the system hardware. The following tasks are performed when this function is called:

1.- The inputs to the measurement system instrumentation amp are disconnected.
2.- The null circuitry is cleared.
3.- The Reference DAC is cleared.
4.- Source R is cleared.
5.- Sources A, B, C, and D are cleared.
6.- Source TH is cleared.
7.- The DDB bits are cleared.
8.- The DCB bits are cleared.
9.- The family board addresses (1800 hex through 19FE) hex are cleared.

Differences with LTS Basic:

None

Example:

```
hdwclr();       /* reset hardware */
```

ltrig

Prototype:

void **ltrig**(void)

Description:

Generates a 100ms , 5 Volts, positive pulse whenever it is called in the test program.

Differences with LTS Basic:

None

Example:

```
dcb(0x20);
ltrig();
meas(1,NOSYNC);
ltrig();
```

lts_printf

Prototype:

int **lts_printf**(char *a,...)

Description:

Allows printing to the test window screen. This function is called in exactly the same way as the standard printf() function is used. Note that the standard printf() function will print to the DOS screen only, and not to the test program window (which is in fact a different video memory page).

Example:

```
lts_printf("\n Result at test number %ld is %3.2f",tnum,res);
```

lts_wait

Prototype:

void **lts_wait**(double a)

Description:

Delay program execution by **a** milliseconds. The PCLTS system has a microsecond resolution timer on the CPU Emulator board. This, together with the very fast execution of the test program instructions, allow for very accurate timing and sequencing. In fact, the user may now generate tightly-controlled pulses of only a few microseconds in width.

This function can set delays ranging from 1 microsecond to 65 milliseconds. For longer delays use the *lts_wait_secs()* function.

Differences with LTS Basic:

IMPORTANT: The LTS Basic WAIT sets a delay specified in seconds. This function sets a delay in milliseconds.

Example:

```
lts_wait(20);          /* wait 20 ms */
```

lts_wait_secs

Prototype:

void **lts_wait_secs**(double a)

Description:

Similar to *lts_wait()* function but allows for delays from 1 millisecond up to 68 years. The parameter passed, **a**, must be in seconds.

Example:

```
lts_wait_secs(0.1);          /* wait 100 ms (0.1 secs) */
```

maxx

Prototype:

void **maxx**(double a)

Description:

Specifies the maximum voltage or current to be measured in the next *meas()* call. The gain of the measurement system is then set to a value such that it does not alarm when making the reading. This function only sets a software variable with the required gain, the actual closing of the gain relays is done when the *meas()* function called. The gain value set will be the maximum possible gain before the system alarm is set.

Differences with LTS Basic:

This function matches the Basic MAX statement. The variable *max_value* replaces the Basic MAX function. Another difference is the spelling. The reason for adding an

extra *x* to the name is to distinguish it from the "max()" function, which is part of the Microsoft C library.

Example:

```
maxx(0.5);          /* set the system gain to measure 0.5V */
```

Important Note:

The programmable gain stage of the measurement system consists of relays that connect parallel combinations of resistors to form the feedback element in a non-inverting op-amp. There is a finite number of resistors and therefore a finite number of possible gain settings. The range of possible gain values spans from 0.8 to 1077.6 and eventhough there is a total of 64 possible values, only 11 make sense being used. This is because gain values are grouped in 11 widely separated clusters.

Please refer to the **Gain Settings** appendix of this manual for more details on gain ranges.

maxd

Prototype:

void **maxd**(double a)

Description:

Specifies the maximum difference in voltage or current to be measured in the next *diff()* call. The gain of the measurement system is then set to a value such that it does not alarm when making the reading. This function only sets a software variable with the required gain, the actual closing of the gain relays is done when the *diff()* function called. The gain value set will be the maximum possible gain before the system alarm is set.

Differences with LTS Basic:

This function matches the Basic MAXD statement. The variable *maxd_value* replaces the Basic MAXD function.

Example:

```
maxd(0.5);          /* set the system gain to measure a difference of 0.5V */
```

Important Note:

The programmable gain stage of the measurement system consists of relays that connect parallel combinations of resistors to form the feedback element in a non-inverting op-amp. There is a finite number of resistors and therefore a finite number of possible gain settings. The range of possible gain values spans from 0.8 to 1077.6 and eventhough there is a total of 64 possible values, only 11 make sense being used. This is because gain values are grouped in 11 widely separated clusters.

Please refer to the **Gain Settings** appendix of this manual for more details on gain ranges.

meas

Prototype:

double **meas**(int a, unsigned short b)

Description:

Performs a measurement **a** times, and returns the average in the *res* system variable. The second parameter, **b**, specifies if synchronization to the power line should be used. To synchronize use SYNC, if not, use NOSYNC.

Differences with LTS Basic:

None, except that the PCLTS *meas()* function always requires two parameters (even if taking only one reading or if no line sync. is requested).

Example:

```
meas(1,NOSYNC);        /* perform a single measurement, no power line sync. */
```

message_to_user

Prototype:

void **message_to_user**(char *a, char *b, char *c, int d)

Description:

This function allows the programmer to display a message to the operator. The message must be of up to three lines and is displayed inside a magenta window opened on top of the test screen. The first three parameters should be pointer to strings containing the three lines displayed on the message. The fourth parameter is a Boolean flag which, if set to TRUE, will cause the computer speaker to "beep" to call the user's attention.

The message will be displayed on the screen until the user hits a key.

Example:

```
char *line1="Device has failed test!";                /* define and initialize strings */
char *line2="Please make sure that the device is ";
char *line3="plugged into the hand socket correctly.";
.
.
.
```

```
message_to_user(line1,line2,line3,TRUE);        /* display message and sound
                                                   beep */
```

msd

Prototype:

void **msd**(double a)

Description:

Specifies the delay in seconds **a** between shutting off the digital bus and strobing the A/D converter in the measurement board.

Important:

On the PCLTS system the CPU board is not present inside the LTS tester case as is the case with the standard LTS 2020 system. The only board that produces digital noise is the CPU emulator, which is inside the PC. For this reason, there is no need to use the *msd()* function. In fact, using it will result in much longer test times and should therefore be avoided. The only reason for providing this function as part of the PCLTS library is because in very high gain tests, like open loop gain in OP-AMPS, it seems to help improve repeatability. These are tests where there is a high gain in both the measurement board and the respective family board. With DACs (even 12-bit) it does not help at all.

Even when using it in a particular test when testing OP-AMPS, *msd()* should be set back to zero after that test is completed in order to optimize the test time. If not set at all in the test program, *msd()* defaults to zero.

Differences with LTS Basic:

This function matches the Basic MSD statement. The variable *msd_value* replaces the Basic MSD function.

Example:

```
msd(0.0001);    /* set delay to 100 us */
```

noise

Prototype:

void **noise**(double a)

Description:

Specify the maximum noise (in Volts) **a** expected when the system measures the null error during a call to the *null()* function. The null error is the voltage left at the measurement system's summing junction after attempting to null that point to zero volts. This remaining small voltage is measured by the system, as part of the *null()*

function call, using the maximum possible gain unless the user specifies a "noise floor" via the *noise()* function. The higher the value specified in *noise()*, the lower the gain setting used to measure the null error will be. An unnecessarily large noise value will therefore cause inaccuracy in the null-diff reading.

Differences with LTS Basic:

This function matches the Basic NOISE statement. The variable *noise_value* replaces the Basic NOISE function.

Example:

```
noise(0.015);              /* set noise value to 15 mV */
```

null

Prototype:

double **null**(int a, unsigned short b)

Description:

Nulls the voltage at the measurement system's summing junction, and measures the null error using either the maximum gain, or the gain corresponding to the value specified in the last *noise()* function call. The first parameter, **a**, specifies the number of readings to average. The second parameter, **b**, specifies if power line synchronization is to be used. Valid second parameters are SYNC and NOSYNC. The function returns the null error (the small voltage remaining at the summing junction after nulling it), and also sets the system variables *res* and *nulerr* to that same value.

The two parameters are only used for measuring the null error. They have no significance for the actual nulling process.

Differences with LTS Basic:

None, except that the PCLTS *null()* function always requires two parameters (even if taking only one reading or if no line sync. is requested).

Example:

```
null(5,NOSYNC);    /* null (measuring null error five times) without line synch. */
```

pclass

Prototype:

void **pclass**(int a, int b)

Description:

Specifies which classes correspond to "passing" devices. The classes specified are classes **a** through **b**, both included. Allowed class values are 1 through 32.

Differences with LTS Basic:

This function is equivalent to "PCLASS a **TO** b " in Basic, but not **"**PCLASS a**,**b". The latter form of the function is not supported. Also, multiple sets of bins may not be listed with a single call of pclass(). So the equivalent of "PCLASS 1,5 TO 7,9" in Basic would be :

```
pclass(1,1); pclass(5,7); pclass(9,9);
```

pulse

Prototype:

void **pulse**(unsigned short a, short b, short c, double d, unsigned short e, short f)

Description:

Allows user to pulse a given bit and, after a specified delay, make a measurement. This function only sets the necessary system variables, the actual pulse and measurement occur during the next *meas()*, *null()*, of *diff()*. The parameters are:

a: base of bit to be pulsed
b: offset of bit to be pulsed
c: pulse polarity (either 0 or 1)
d: delay, in seconds, between the trailing edge of the pulse and measurement
e: base of optional second pulsed bit
f: offset of optional second pulsed bit

To turn the pulse option off, the function should be called with the following parameters:

```
pulse(OFF,0,0,0,0,0);
```

When not using the optional second pulsed bit, call the function with the last two parameters set to zero, as follows:

```
pulse(0x1800, 3,1,0.004,0,0);
```

Differences with LTS Basic:

Syntax differences are as described above. Another difference is that the PCLTS pulse function allows for a smaller minimum delay of 30us (as opposed to 80us on the LTS Basic version). The maximum is also 20ms

rdcode

Prototype:

void **rdcode**(unsigned short a)

Description:

Writes the specified 12-bit code to the reference DAC. Valid codes are 0 to 4095, both included. The resulting output voltage of the reference DAC will depend on the mode (gain and polarity) set by the last *span()* function call.

Differences with LTS Basic:

This function matches the Basic RDCODE statement. The variable *rdcode_value* replaces the Basic RDCODE function.

Example:

```
rdcode(0x125);          /* set the code of the ref DAC to 125 hex */
```

read_family_board_id

Prototype:

unsigned char **read_family_board_id**(void)

Description:

Returns the family board ID. All LTS family boards have a factory 8-bit hard-wired code that distinguishes from another.

Example:

```
code=read_family_board_id();          /* read ID */
if(code!=14){
  lts_printf("\n This program must use the 2302 DAC family board only!!");
}
```

record_data_point

Prototype:

int **record_data_point**(double a)

Description:

This function will record a data point for use with an external graphical or mathematical analysis program. Data points are dynamically stored in a linked list in memory, and are then written to a target file once the execution of the test program is completed.

Actual data collection does not occur until activated by the user by means of the *Collect Data* option under the *Engineering* menu.

Example:

```
meas(1,NOSYNC);
inl_value=(res-(vzs+code*lsb))/lsb;        /* calculate result */
record_data_point(inl_value);              /* record that new point */
```

The recorded values in the example could then be used by DADiSP or another graphical analysis program to generate a DAC linearity plot.

run_time_error_msg_to_user

Prototype:

void **run_time_error_msg_to_user**(char *a, char *b, char *c, int d)

Description:

This function allows the programmer to display an error message to the operator. Upon execution of this function, the system will generate a run-time error and the program will be aborted. As a result, snum will not be incremented and the datalog data (if dlog is on) will not be dumped. The error message must be of up to three lines and is displayed inside a red window opened on top of the test screen. The first three parameters should be pointer to strings containing the three lines displayed on the message. The fourth parameter is a Boolean flag which, if set to TRUE, will cause the computer speaker to "beep" to call the user's attention.

The message will be displayed on the screen until the user hits a key.

Example:

```
char *line1="Device has failed a critical test!";      /* define and initialize strings */
char *line2="Please call test technician.";
char *line3="Program will be aborted!!";
.
.
run_time_error_msg_to_user(line1,line2,line3,TRUE);   /* display msg., sound
                                                         beep */
```

select

Prototype:

void **select**(unsigned short a, unsigned short b)

Description:

Connects the specified lines to the inputs of the measurement system's instrumentation amplifier. Parameter **a** corresponds to the positive input, and parameter **b** corresponds to the negative input.

Differences with LTS Basic:

None

Example:

```
select(LA,GD);    /* connect LA to the + input and GD to the negative input */
```

set_gain_relays

Prototype:

void **set_gain_relays**(unsigned short a)

Description:

Closes the measurement system gain relays that correspond to the last *maxx()* or *maxd()* function call. It does not wait for the relays to settle. The user must add an *lts_wait()* function call after closing the relays, if needed.

This function is used together with *fast_meas()* and *fast_diff(),* and it replaces the LTS Basic CALL Mx subroutine calls.

The parameter passed to this function should be **MEASX** if used with *fast_meas()* or **DIFFX** if used with *fast_diff()*.

Example:

```
maxx(2.4);
set_gain_relays(MEASX);  /* These 2 lines replace the LTS Basic "CALL M4"  */
fast_meas();

maxd(2.4);
set_gain_relays(DIFFX);  /* These 2 lines replace the LTS Basic "CALL M4"  */
fast_diff();
```

span

Prototype:

void **span**(unsigned short a, unsigned short b)

Description:

Specifies the span and offset of the Reference DAC. The **a** parameter corresponds to the span and can have a value of either 10 or 20. The second parameter, **b**, corresponds to the offset, and can have be either UPO (for unipolar) or BPO (for bipolar). This function only specifies the mode of the Reference DAC. In order to set an actual voltage, the *vrd()* or *rdcode()* functions must be used.

The possible settings are:

```
span(10,UPO)   for 0V to 10V range
span(10,BPO)   for -5V to +5V range
span(20,UPO)   for 0V to 20V range (limited to approx. 12V )
span(20,BPO)   for -10V to +10V range
```

Differences with LTS Basic:

None

Example:

```
span(10,BPO);          /* set Ref DAC to -5V to +5V range */
vrd(2.3);              /* set Ref DAC to 2.3 Volts */
```

srcode

Prototype:

void **srcode**(unsigned short a)

Description:

Writes the specified 16-bit code to source R DAC. Valid codes are 0 to 65535, both included.

Differences with LTS Basic:

This function matches the Basic SRCODE statement. The variable *srcode_value* replaces the Basic SRCODE function.

Example:

```
srcode(0x12fb);        /* set the code of source R DAC to 12FB hex */
```

sys_reset

Prototype:

void **sys_reset**(void)

Description:

Will reset the entire LTS system hardware. This function pulses the IORST line on the LTS I/O bus. This will reset all the measurement system hardware, like *hdwclr()* does, but with the difference that *sys_reset()* will also clear the information programmed into the handler board (interface timing and voltage levels of the handler port).

Example:

```
sys_reset();
```

test

Prototype:

int **test**(void)

Description:

Performs the classification of the device under test. It performs the following tasks:

1.- If the system variable *fail* is set, eliminate the classes specified in the last *fclass()* function call.
2.- If the system variable *alarm* is set, eliminate the classes specified in the last *aclass()* function call. If the last *aclass()* function had zeroes as parameters (or if *aclass()* has not been called so far) then eliminate the classes in the last *fclass()* function call.
3.- Clear the *fail* and *alarm* variables.
4.- If all the pass classes have been eliminated, the *device_failed* system variable is set to TRUE.

Differences with LTS Basic:

None. Of course the line number option of the LTS Basic TEST statement has no meaning in the PCLTS system.

Example:

```
test();
```

test_parameters

Prototype:

int **test_parameters**(double a, char *b, char *c, double d, double e, char *f, int g)

Description:

Sets the necessary parameters for the current test. The **a** parameter must be the result of the current test; **b** must be a pointer to a string with the test name; **c** must be a pointer to a string containing the units; **d** and **e** must be the lower and upper limits, respectively; **f** must be a pointer to a string with the number formatting decimals to be used when data-logging; and **g** must be the fail bin. This function should only be called when a limit table is not being used. Use *auto_test_parameters_and_bin()* when a limit table is used.

Example:

```
meas(1,NOSYNC);                                    /* measure */
test_parameters(res,"Idd", "mA", 0.01,25, "3",9);  /* set test parameters */
bin_current_test(res);                             /* bin this test */
```

vsa

Prototype:

void **vsa**(double a)

Description:

Sets the voltage of source A to **a** volts. The voltage requested must be between 0 Volts and 255*(lsb of VSA DAC).

Differences with LTS Basic:

This function is identical to the Basic VSA statement. The variable *vsa_value* replaces the Basic VSA function.

Example:

```
vsa(5.3);          /* set source A to 5.3 Volts */
```

vsb

Prototype:

void **vsb**(double a)

Description:

Sets the voltage of source B to **a** volts. The voltage requested must be between 0 Volts and 255*(lsb of VSB DAC).

Differences with LTS Basic:

This function is identical to the Basic VSB statement. The variable *vsb_value* replaces the Basic VSB function.

Example:

```
vsb(3.4);          /* set source B to 3.4 Volts */
```

vsc

Prototype:

void **vsc**(double a)

Description:

Sets the voltage of source C to **a** volts. The voltage requested must be between 255*(lsb of VSC DAC) and 0 Volts

Differences with LTS Basic:

This function is identical to the Basic VSC statement. The variable *vsc_value* replaces the Basic VSC function.

Example:

```
vsc(2.1);          /* set source C to 2.1 Volts */
```

vsd

Prototype:

void **vsd**(double a)

Description:

Sets the voltage of source D to **a** volts. The voltage requested must be between 255*(lsb of VSD DAC) and 0 Volts

Differences with LTS Basic:

This function is identical to the Basic VSD statement. The variable *vsd_value* replaces the Basic VSD function.

Example:

```
vsd(-8.2);         /* set source D to -8.2 Volts */
```

vsr

Prototype:

void **vsr**(double a)

Description:

Sets the voltage of source R to **a** volts. The voltage requested must be between the offset voltage of source R and the offset voltage plus the sum of the contribution of all its bits.

Differences with LTS Basic:

This function is identical to the Basic VSR statement. The variable *vsr_value* replaces the Basic VSR function.

Example:

```
vsr(-2.005);        /* set source R to -2.005 Volts */
```

vrd

Prototype:

void **vrd**(double a)

Description:

Sets the voltage of the Reference DAC. The requested voltage **a** must be within the range specified in the last *span()* function call.

Differences with LTS Basic:

This function is identical to the Basic VRD statement. The variable *vrd_value* replaces the Basic VRD function.

Example:

```
vrd(6.23);                  /* set the Ref DAC to 6.23 Volts */
```

vth

Prototype:

void **vth**(double a)

Description:

Sets the voltage of the threshold DAC on the digital I/O board to the value specified by **a**.

Differences with LTS Basic:

This function is identical to the Basic VTH statement. The variable *vth_value* replaces the Basic VTH function.

Example:

```
vth(2.3);
```

PCLTS System Variables

The following is a listing of all the PCLTS system variables. These are global variables that the user may use at any point in the test program. These have already been declared, so they should not be declared again in the test program source files.

Variables Listing

The following is a comprehensive listing of the PCLTS system variables. For each variable, it includes the C-language type, a description, the differences between the original LTS Basic function and the PCLTS version of it, and if it can be safely modified (assigned a value) by the user. A usage example is also included for further clarification.

alarm

Type:

int **alarm**

Description:

Will be set to TRUE if the measurement system over-ranged during the last *meas()*, *fast_meas()*, *diff()*, or *fast_diff()* function calls.

Differences with LTS Basic:

None..

Can be safely modified by the user:

Yes. However, the user should only clear this variable, not set it.

Example:

```
if(alarm) alarm=0;          /* if alarm is set, clear it */
```

base_value

Type:

unsigned short **base_value**

Description:

Keeps the value of the base address set by the last *base()* function call.

Differences with LTS Basic:

Not available in LTS Basic.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

The main purpose of this variable is to "watch" it from Codeview, but it may also be used in the test program:

```
lts_printf("\n The current CRU base address is: %d",base_value);
```

aclasv

Type:

unsigned long **aclasv**

Description:

Keeps the value of the classification system data for the alarm condition. It is a 32-bit integer, and its bits represent the 32 classes (bins).

Differences with LTS Basic:

The only difference is that the PCLTS *aclasv* variable represents 32 classes instead of 48. It replaces both the Basic ACLASV function and statement.

Can be safely modified by the user:

Yes.

Example:

Used as the Basic ACLASV function:

```
x=aclasv;
```

Used as the Basic ACLASV statement:

```
aclasv=0x01;
```

adrdg

Type:

double **adrdg**

Description:

Keeps the value of the measurement system's A/D converter last reading.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(adrdg==0){
  lts_printf("\n Meter is pegged low...");
}
```

advin

Type:

double **advin**

Description:

Keeps the calculated value of the voltage present at the input of the measurement system's A/D converter.

Differences with LTS Basic:

None

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The value of ADVIN is : %f",advin);
```

bin_descriptions[ ]

Type:

char ***bin_descriptions[ ]**

Description:

This array of strings (array of pointers to characters) contains the name of all the 32 bins on the device classification system. They are used for the summary report's "Device Distribution by Bin" table. The user must assign the respective names in the *define_bin_descriptions()* function in the test program source file.

Note that this is an array of pointers and not an array of character arrays. The strcpy() function may therefore not be used to assign the bin names. Instead, immediate pointer assignments can be used:

```
either
        bin_descriptions[12]="Supply Current";

or
        char *name="Supply Current";     /* declare and initialize the string */
        ...
        bin_descriptions[12]=name;       /* assign the string (pointer) */
```

Differences with LTS Basic:

N/A

Can be safely modified by the user:

Yes.

Example:

See above.

clasv

Type:

unsigned long **clasv**

Description:

Keeps the value of the device classification system data. It is a 32-bit integer, and its bits represent the 32 possible classes (bins).

Differences with LTS Basic:

The only difference is that the PCLTS *clasv* variable represents 32 classes instead of 48. It replaces both the Basic CLASV function and statement.

Can be safely modified by the user:

Yes.

Example:

Used as the Basic CLASV function:

```
if(aclasv==0xfffffffe){     /* if only class 0 has been disqualified so far */
        part_is_a_bin_2=TRUE;
}
```

Used as the Basic CLASV statement:

```
aclasv=aclasv & 0x0fffffffc;   /* disqualify classes 1 and 2 */
```

Note: In the example above, the *disqualify_bin()* function could be used instead.

cmv_value

Type:

double **cmv_value**

Description:

Keeps the value specified in the last *cmv()* function call.

Differences with LTS Basic:

Equivalent to the LTS Basic CMV function (not statement).

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=cmv_value;
```

dcb_value

Type:

short int **dcb_value**

Description:

Keeps the value specified in the last *dcb()* function call.

Differences with LTS Basic:

Equivalent to the LTS Basic DCB function (not statement).

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
dcb(dcb_value+1);   /* equivalent to LTS Basic DCB=DCB+1 */
```

ddb_value

Type:

short int **ddb_value**

Description:

Keeps the value specified in the last *ddb()* function call.

Differences with LTS Basic:

Equivalent to the LTS Basic DDB function (not statement).

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
ddb(ddb_value | 0x3);   /* equivalent to LTS Basic DDB=DDB LOR 03H */
```

device_failed

Type:

int **device_failed**

Description:

Will have a TRUE (non-zero) value if the DUT has failed the test program.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(device_failed){
  lts_printf("\n The DUT has failed the test program!");
}
```

difflg

Type:

int **difflg**

Description:

Allows user to determine if the last measurement was a *diff().* If that is the case, it will be set to a logic TRUE (non-zero) value.

Differences with LTS Basic:

None

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(difflg){
  lts_printf("\n The last measurement was a DIFF...");
}
```

fclasv

Type:

unsigned long **fclasv**

Description:

Keeps the value of the classification system data for the fail condition. It is a 32-bit integer, and its bits represent the 32 classes (bins).

Differences with LTS Basic:

The only difference is that the PCLTS *fclasv* variable represents 32 classes instead of 48. It replaces both the Basic FCLASV function and statement.

Can be safely modified by the user:

Yes.

Example:

Used as the Basic FCLASV function:

```
x=fclasv;
```

Used as the Basic FCLASV statement:

```
fclasv=0x03;
```

folvin

Type:

double **folvin**

Description:

Keeps the value of the measurement system's voltage follower input voltage.

Differences with LTS Basic:

None

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The value of folvin is %f",folvin);
```

gbgain

Type:

double **gbgain**

Description:

Keeps the value of the measurement system's gain buffer gain setting.

Differences with LTS Basic:

None

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The current gain of the system is %f",gbgain);
```

hiflg

Type:

int **hiflg**

Description:

Keeps the status of the handler port. Will be TRUE (non-zero value) if the handler port is currently enabled.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(hiflg){
  x=5;
}
```

initial_run

Type:

int **initial_run**

Description:

Will have a TRUE (non-zero) value the first time that the program is run. After testing the first device, the system sets this variable to FALSE.

Differences with LTS Basic:

Replaces the LTS Basic STAT function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(initial_run){
  verify_family_board();   /* read family board ID & make sure it's the proper ID */
}
```

limits_type[ ]

Type:

char **limits_type[ ]**

Description:

This string (array of characters) stores the name of the limits type entered by the operator on the Operator Data Entry window the first time that the program is run. If no limits table is being used, then this string will be empty. It can be used in the program to change the flow or conditions of the testing depending on what type of limits are loaded.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(strcmpi(limits_type,"QA")==0){   /* if QA limits were loaded... */
  ...
}
```

lsync

Type:

int **lsync**

Description:

Will be TRUE (non-zero value) if the last measurement was synchronized to the power line.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(lsync)  lts_printf("\n The last measurement was line sync'd");
```

maxgbv

Type:

double **maxgbv**

Description:

Keeps the value of the maximum expected voltage at the measurement system's gain buffer. This value is calculated from the last *maxd()* or *maxx()* function call.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The value of MAXGBV is %f",maxgbv);
```

maxd_value

Type:

double **maxd_value**

Description:

Keeps the value of the maximum expected voltage or current to be measured, as specified on the last *maxd()* function call.

Differences with LTS Basic:

Replaces the LTS Basic MAXD function, not statement.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The last is maxd() specified a value of %f",maxd_value);
```

max_value

Type:

double **max_value**

Description:

Keeps the value of the maximum expected voltage or current to be measured, as specified on the last *maxx()* function call.

Differences with LTS Basic:

Replaces the LTS Basic MAX function, not statement.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The last is maxx() specified a value of %f",max_value);
```

nin

Type:

int **nin**

Description:

Keeps the value of the line connected to the negative input of the measurement system instrumentation amplifier. This value is specified in the second parameter of the last *select()* function call. The values assigned have been kept identical to the ones used by the LTS Basic system. These values are:

LC	4096
LD	8192
GD	512
RD	2048
TN	1024
CUR	16384
VLTG	32768

Differences with LTS Basic:

None.

Can be safely modified by the user:

No. If these values are modified, the entire measurement system will be corrupted! These values should never be modified. (The reason for not using constants instead of variables here, is that constants do not retain its symbolic information for Codeview, and it would not have been possible to execute instructions like "select(LA,LC)" from the Codeview command window.)

Example:

```
if(nin==LC){          /*  if we used select(?,LC)   */
  maxx(0.1);
}
```

noise_value

Type:

double **noise_value**

Description:

Keeps the value specified in the last *noise()* function call.

Differences with LTS Basic:

Equivalent to the LTS Basic NOISE function (not statement).

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=noise_value;
```

nulerr

Type:

double **nulerr**

Description:

Keeps the value of the *res* variable after a *null()* call. This value corresponds to the small voltage that remains at the measurement system's summing junction after the system tries to "null" that junction to zero volts.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

The main purpose of this variable is to "watch" it from Codeview, but it may also be used in the test program:

```
x=nulerr;
```

nulflg

Type:

int **nulflg**

Description:

Will have a TRUE (non-zero) value if the last measurement was a *null()*.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(nulflg){
  lts_printf("\n The last measurement was a null...");
}
```

numavg

Type:

int **numavg**

Description:

Keeps the number of averages specified if the last measurement.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
lts_printf("\n The last measurement performed %d readings..",numavg);
```

pin

Type:

int **pin**

Description:

Keeps the value of the line connected to the positive input of the measurement system instrumentation amplifier. This value is specified in the first parameter of the last *select()* function call. The values assigned have been kept identical to the ones used by the LTS Basic system. These values are:

LA	2
LB	4
SA	8
SB	16
SC	32
SD	64
SR	128
TH	256
GD	512
RD	2048
TN	1024

Differences with LTS Basic:

None.

Can be safely modified by the user:

No. If these values are modified, the entire measurement system will be corrupted! These values should never be modified. (The reason for not using constants instead of variables here, is that constants do not retain its symbolic information for Codeview, and it would not have been possible to execute instructions like "select(LA,LC)" from the Codeview command window.)

Example:

```
if(pin==LA){          /*  if we used select(LA,?)   */
  maxx(1);
}
```

product_name[ ]

Type:

char **product_name[ ]**

Description:

This string (array of characters) stores the name of the product being tested. It must be set by the user in the test program by defining the PRODUCT_ID constant.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

Yes.

Example:

```
lts_printf("\n The product being tested is :%s",product_name);
```

program_run_by_Verify_Setup

Type:

int **program_run_by_Verify_Setup**

Description:

Will have a TRUE (non-zero) value if the test program execution was initiated by the "Verify Setup" option of the LTSSHEL program, or if it was initiated by the CORRLGEN program "Create Trial File" option. This variable may be used in the program to change the flow or conditions of the testing when running a setup verification.

Differences with LTS Basic:

Not available in LTS Basic.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(!program_run_by_Verify_Setup){   /* skip this during setup verification */
  ...
}
```

res

Type:

double **res**

Description:

Keeps the result of the last measurement.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
result=res*1000;                    /* convert to mV */
```

rdvo

Type:

double **rdvo**

Description:

Keeps the calculated value of the Reference DAC's output voltage, as opposed to the value requested in the last *vrd()* function call (which is kept in *vrd_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=rdvo;
```

rdcode_value

Type:

unsigned short **rdcode_value**

Description:

Keeps the value of the 12-bit code to which the Reference DAC was set in the last *vrd()* or *rdcode()* function call.

Differences with LTS Basic:

Replaces the LTS Basic RDCODE function, not statement.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=rdcode_value;
```

rtstfl

Type:

int **rtstfl**

Description:

Keeps the voltage value specified for the Reference DAC in the last *vrd()* or *rdcode()* function call. The calculated voltage value is stored in the *rdvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vrd(vrd_value+0.05);          /* increment the current voltage by 50 mV */
```

savo

Type:

double **savo**

Description:

Keeps the calculated value of source A's output voltage, as opposed to the value requested in the last *vsa()* function call (which is kept by *vsa_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=savo;
```

SA_is_on

Type:

int **SA_is_on**

Description:

Will have a TRUE (non-zero) value source A was gated on.

Differences with LTS Basic:

Replaces the LTS Basic GATE function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(SA_is_on){
  lts_printf("\n  Source A is on");
}
```

sbvo

Type:

double **sbvo**

Description:

Keeps the calculated value of source B's output voltage, as opposed to the value requested in the last *vsb()* function call (which is kept by *vsb_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=sbvo;
```

SB_is_on

Type:

int **SB_is_on**

Description:

Will have a TRUE (non-zero) value source B was gated on.

Differences with LTS Basic:

Replaces the LTS Basic GATE function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(SB_is_on){
  lts_printf("\n  Source B is on");
}
```

scvo

Type:

double **scvo**

Description:

Keeps the calculated value of source C's output voltage, as opposed to the value requested in the last *vsc()* function call (which is kept by *vsc_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=scvo;
```

SC_is_on

Type:

int **SC_is_on**

Description:

Will have a TRUE (non-zero) value source C was gated on.

Differences with LTS Basic:

Replaces the LTS Basic GATE function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(SC_is_on){
  lts_printf("\n  Source C is on");
}
```

sdvo

Type:

double **sdvo**

Description:

Keeps the calculated value of source D's output voltage, as opposed to the value requested in the last *vsd()* function call (which is kept by *vsd_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=sdvo;
```

SD_is_on

Type:

int **SD_is_on**

Description:

Will have a TRUE (non-zero) value source D was gated on.

Differences with LTS Basic:

Replaces the LTS Basic GATE function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(SD_is_on){
  lts_printf("\n  Source D is on");
}
```

snum

Type:

long int **snum**

Description:

Keeps the serial number of the current device.

Differences with LTS Basic:

None.

Can be safely modified by the user:

Yes. However, it does not make sense to modify this variable from inside the test program.

Example:

```
lts_printf("\n  Current serial number is %d",snum);
```

span_value

Type:

int **span_value**

Description:

Keeps the value of the Reference DAC voltage span that was set in the last *span()* function call. The possible values are:

0	Hex for 20,UPO
100	Hex for 20,BPO
200	Hex for 10,UPO
300	Hex for 10,BPO

Differences with LTS Basic:

Replaces the LTS Basic SPAN function, not statement. This function is not documented on the CTS LTS-2020 software manual, but it exists.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(span_value==0x300){
  lts_printf("\n Ref DAC span was set to 10,BPO");
};
```

srcode_value

Type:

unsigned short **srcode_value**

Description:

Keeps the value of the code to which source R was set in the last *vsr()* or *srcode()* function call.

Differences with LTS Basic:

Replaces the LTS Basic SRCODE function, not statement.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=srcode_value;
```

SR_is_on

Type:

int **SR_is_on**

Description:

Will have a TRUE (non-zero) value source R was gated on.

Differences with LTS Basic:

Replaces the LTS Basic GATE function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(SR_is_on){
  lts_printf("\n  Source R is on");
}
```

sys_calibration_ocurred

Type:

int **sys_calibration_ocurred**

Description:

Will have a TRUE (non-zero) value after the system performs its hourly calibration. This flag can be used in the test program to perform periodic tasks, like calibration of the family board being used.

After testing the device following the calibration, the system sets this variable to FALSE.

Differences with LTS Basic:

Replaces the LTS Basic CALFLG function.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(sys_calibration_ocurred){     /* calibrate fam. board hourly, after system cal. */
  calibrate_family_board();
}
```

test_downgraded

Type:

int **test_downgraded**

Description:

Will have a TRUE (non-zero) value if the DUT has downgraded on the current test. This will happen if class 1 has been disqualified but there are still valid pass classes remaining which have not been disqualified.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(test_downgraded){
  lts_printf("\n The DUT has downgraded");
}
```

test_conditions_code[ ]

Type:

char **test_conditions_code[ ]**

Description:

This string (array of characters) stores the test conditions code entered by the operator on the Operator Data Entry window the first time that the program is run. This variable has been included to comply with the STDF specification and can be used in the program to change the flow or conditions of the testing. Its use is optional.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(strcmpi(test_conditions_code,"883")==0){   /* if code is 883 then.. */
  ...
}
```

test_mode_code[ ]

Type:

char **test_mode_code[ ]**

Description:

This string (array of characters) stores the test mode code entered by the operator on the Operator Data Entry window the first time that the program is run. This variable has been included to comply with the STDF specification and can be used in the program to change the flow or conditions of the testing. Its use is optional.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
if(strcmpi(test_mode_code,"B")==0){   /* if code is B then.. */
  ...
}
```

thvo

Type:

double **thvo**

Description:

Keeps the calculated value of the threshold DAC's output voltage, as opposed to the value requested in the last *vth()* function call (which is kept by *vth_value*). These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
x=thvo;
```

tnum

Type:

long int **tnum**

Description:

Keeps the current test number value. This system variable is automatically incremented by the system when the user calls either *bin_current_test()* (if not using a limit table) or *auto_test_parameters_and_bin()* (if a limit table is being used). Tnum is initially set to 1 by the system at the start of the test program.

This variable can be set to a given value at any point in the test program. An example of this would be if the user was using a limit table and wanted to skip some tests.

Differences with LTS Basic:

None.

Can be safely modified by the user:

Yes.

Example:

```
tnum=40;     /* reset test number */
```

user_switch_1, user_switch_2, user_switch_3

Type:

int **user_switch_1, user_switch_2, user_switch_3**

Description:

These three flags have been included for added program flexibility. They can easily set by the operator from the "Options" menu and can be used in the program to change the flow or conditions of the testing. Their status can be checked at any time during program execution in the SETUP window (F7). Its use is optional.

Differences with LTS Basic:

N/A

Can be safely modified by the user:

They are modified at run time by the operator.

Example:

```
if(user_switch_1){
  vsa(15);
}
else{
  vsa(12);
}
```

vrd_value

Type:

double **vrd_value**

Description:

Keeps the voltage value specified for the Reference DAC in the last *vrd()* or *rdcode()* function call. The calculated voltage value is stored in the *rdvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vrd(vrd_value+0.05);          /* increment the current voltage by 50 mV */
```

vsa_value

Type:

double **vsa_value**

Description:

Keeps the voltage value specified for source A in the last *vsa()* function call. The calculated voltage value is stored in the *savo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vsa(vsa_value+0.05);          /* increment the current voltage by 50 mV */
```

vsb_value

Type:

double **vsb_value**

Description:

Keeps the voltage value specified for source B in the last *vsb()* function call. The calculated voltage value is stored in the *sbvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vsb(vsb_value+0.05);          /* increment the current voltage by 50 mV */
```

vsc_value

Type:

double **vsc_value**

Description:

Keeps the voltage value specified for source C in the last *vsc()* function call. The calculated voltage value is stored in the *scvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vsc(vsc_value+0.05);          /* increment the current voltage by 50 mV */
```

vsd_value

Type:

double **vsd_value**

Description:

Keeps the voltage value specified for source D in the last *vsd()* function call. The calculated voltage value is stored in the *sdvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vsd(vsd_value+0.05);          /* increment the current voltage by 50 mV */
```

vsr_value

Type:

double **vsr_value**

Description:

Keeps the voltage value specified for source R in the last *vsr()* function call. The calculated voltage value is stored in the *srvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vsr(vsr_value+0.05);          /* increment the current voltage by 50 mV */
```

vth_value

Type:

double **vth_value**

Description:

Keeps the voltage value specified for the threshold DAC in the last *vth()* function call for. The calculated voltage value is stored in the *thvo* variable. These two values are different because of the discrete resolution of the converter.

Differences with LTS Basic:

None.

Can be safely modified by the user:

No, or else it will be meaningless.

Example:

```
vth(vth_value+0.05);          /* increment the current voltage by 50 mV */
```

Troubleshooting Hardware

The DIAGNOSE.EXE Utility

The PCLTS system has a built-in hardware diagnostics utility. This utility, DIAGNOSE.EXE, can be accessed through the ***Run Board Diagnostics*** option (option 8) of the LTSSHEL program menu. The user has the option of executing any of five different tests, or all of them. These tests check the integrity of key sections of both the CPU Emulator and Bus Driver boards.

Diagnostics Utility Menu

Auxiliary Register Test

The auxiliary register is a key hardware register found on the CPU Emulator board inside the PC. It is written and read by the system hardware and software, and is essential for proper operation. If it malfunctions, the system will not operate properly.

This test writes and reads back several bit patterns and checks that the data read is the proper one.

CRU Write Test

The CRU is the block within the original LTS-2020 CPU (the TMS-9900) that handles the serial I/O. Its function is performed by the CPU emulator in the PCLTS system.

In this test, the CPU emulator writes a series of patterns to LEDs on the bus driver card while simultaneously displaying them on the screen. The user must visually

check that the pattern displayed on the bus driver board's LEDs matches the one on the screen.

This test checks the CPU emulator board and the interconnecting cables. If it fails, make sure that the CPU emulator board is fully seated on the PC's I/O bus connectors, and that all four 25-pin connectors on the two interconnecting cables are tightly plugged on both the PC and LTS console ends.

The *CRU Write Test* Screen

CRU Read Test

This test writes and reads 256 codes (codes 0 through 255) to the bus driver board. This test may fail due to either a write failure, read failure, or both. If the CRU read test above passed, then the system is failing to write properly. Failed codes are displayed on the screen.

This test checks the CPU emulator board, the bus driver board and the interconnecting cables. If it fails, make sure that the CPU emulator board is fully seated on the PC's I/O bus connectors, and that all four 25-pin connectors on the two interconnecting cables are tightly plugged on both the PC and LTS console ends. Also check that the bus driver board is properly seated on the LTS console's backplane. Remember that this board must be plugged only in the connector that the original LTS CPU card was occupying. This is the connector that is exactly in the middle of the backplane, with four connectors in front of it and four in back of it. It is also the one that was has the rails marked (from step 6 of the **Installing the Bus Driver Board** section).

Timer Test

The CPU Emulator board contains an on-board programmable interval timer chip (Intel's P8253) which provides the system with a 1us-resolution wait function (namely *lts_wait*). This timer is totally independent of the PC's real-time chip, which are normally inaccurate by a significant amount and certainly do not have microsecond resolution.

The timer is not checked for accuracy (it has 1 microsecond resolution) but, rather, for functionality. Since it uses a low tolerance 1 MHz crystal oscillator, accuracy is guaranteed. The check is done by waiting 2 million microseconds (2 secs) and comparing this interval to 2 seconds of the PC's real-time clock. It is not an accuracy check (since the PC is not as accurate), but rather a functionality check.

Note:

Do not run this test from inside a graphical DOS window in Microsoft Windows, or it will fail. If it is run from Windows, it must be run from a full DOS text screen.

Line Synch Test

The power line synchronization circuitry is on the Bus Driver board inside the LTS console. The circuit detects the zero-crossing of the AC power line and synchronizes measurements to that point in time thus minimizing the interference of the line on the measurement.

This test checks for the occurrence of around 2 seconds worth of zero-crossings of the power line and compares it to 2 seconds of the PC's real time clock. The limits used for this test are fairly loose since it must work for both 50 and 60 Hz power lines worldwide. Since this test is only a functionality check of the line-synch circuitry, the loose limits are not a problem.

If this test fails, it is probably due to a defective U4, D1, D2, or R1 on the bus driver board.

Faulty Serial Port Ribbon Cables Inside LTS

The two 25-pin connectors on the back of the LTS tester are connected to two 26-lead ribbon cables which end in 26-pin IDC connectors inside (these two cables should have been labeled "A" and "B" in step 7 of the section labeled **Installing the Bus Driver Board** in the **Hardware Installation** chapter). Several of these cables have been found to have unreliable connections where the DB-25 connector crimps the ribbon cable. These faulty contacts may have gone unnoticed when the LTS was operated in its original mode because in the serial port not all lines are used and because the noise margin of the RS-232 standard is much higher (due to the use of 12-volt signals). With this extra noise margin, contacts with several tens of ohms could still operate properly.

In the PCLTS system, however the two DB-25 connectors are not used as serial ports, but instead, to bring the CPU emulator board signals to the LTS bus. It is therefore crucial that all lines have continuity since all of them are used. Also, signals are 5-volt TTL, which have significantly less noise margin than 12-volt RS-232.

If these cables have open lines or more than a fraction of an Ohm from one end to the other, system operation may be erratic or totally non-functional.

Please refer to appendix D, Interconnecting Cables, for a full description of these cables.

Tighten Cable Connectors

The connectors on the two 25-lead cables connecting the LTS console with the CPU Emulator in the PC should be tightly secured and have a good quality electrical connection at both the PC and LTS ends. This is crucial since these cables carry all the signals to the LTS backplane, including a 4MHz CPU clock signal.

Do not replace the cables provided with the system for others with inferior quality or much longer ones. If the ones provided break down, please purchase similar cables at your local computer dealer or contact Calyx for a new pair.

Troubleshooting System Software

Codeview Screen "Hangs-up" When Debugging

When a program is being debugged using Microsoft's Codeview debugger, there are two screens in two separate video pages: one is Codeview's user interface screen, where the program's source code can be seen, and the other is the screen of the program being debugged, in this case, the test program main screen. Every time the user single-steps through lines of code, Codeview momentarily switches to the test program's main screen and then back to Codeview's screen again, causing a rather annoying flicker. If the **Screen Swap** mode is turned off (using Codeview's Options menu), only Codeview's screen with the source code will be seen when single-stepping and the flickering will go away.

However, a confusing situation will arise when the test program is executed until the end. At the moment the *test_program()* function returns to the *main()* function (which is part of the system), the test program will be back to its main screen, waiting for the operator to either make a menu selection or hit TEST. If the Screen Swap mode was still on, the screen would show the test program's main screen at this point, but since it is turned off, the user will still see the Codeview screen, even though it is the test program that is in control. When this happens, the screen will appear to be in a "hung-up" state since it will not respond to the mouse. A good way to test if the test program is in control at that moment is to hit F7. If the Current Setup pop-up window appears, then the test program is in control and all the user needs to do to execute the *test_program()* function again is to hit F1 (TEST).

When Codeview Really Hangs-up

Ocassionally, Codeview may become locked when debugging a test program. Sometimes when this happens, a series of error messages will scroll through the command window indicating a floating point error. When this happens, we have found that deleting the CURRENT.STS file will solve the problem. This file is a Microsoft system file and can be found in the INIT sub-directory of the C-compiler directory. It contains all the menu options selected by the user during the last debugging session. For more details please refer to the **The CURRENT.STS File** section in the **Debugging the Test Program** chapter of this manual.

When this file is deleted, Codeview will create a new one upon exiting the next debugging session.

Not Enough Available Memory

The PCLTS system requires over 600K of available DOS memory to ensure that very large test programs (over 300 tests) will run properly. There is a large amount of DOS memory that is dynamically allocated to the test program as it runs. The amount of available DOS memory may be checked at any time by using the

Engineering menu on the test program pull-down menu (user must be logged with engineer privilege level by typing the "/P1" switch at the DOS command line). The memory available should not drop under 100k or else the system will generate an error. This will probably occur when the system runs the hourly system calibration. The error message "*System cannot launch child process...*" will appear on the screen.

TSRs and Device Drivers Should be Loaded High

All memory-resident programs (device drivers and TSRs such as the mouse driver, disk cache and others) should be installed in "high" memory. High memory is the term used to denote RAM memory between 640K and 1 Megabyte. By loading memory-resident programs in high memory, the available (free) DOS memory space is maximized (first 640K of RAM).

Use an Automated Utility to Load Drivers High

There are several DOS utility programs commercially available that may be used to examine the PCs configuration and automatically load memory-resident programs in high memory. Two examples of these are:

- 386MAX by Qualitas Inc.
- QEMM by QuarterDeck Inc.

386MAX includes the MAXIMIZE utility and QEMM has the OPTIMIZE utility. Both of these provide automated loading of memory-resident programs in high memory.

Some PCNFS Drivers May Not be Loaded High

If Sun Microsystem's PCNFS is being used for networking, be aware that some of the PCNFS device drivers may not be loaded "high". Please refer to SUN's documentation for more details.

"VRD Out of Range" Error

The Reference DAC on the LTS measure board is software-calibrated every hour by the system software. As a result of this calibration, the system knows the maximum voltage that this DAC can deliver. The LTS Basic ADOS system for some reason does not check this voltage properly. It actually allows the user to set the DAC to a value somewhat larger than the maximum it can deliver, without displaying an error (until the user tries to set it to a voltage significantly larger than the maximum possible).

The practical effect of all this is that some LTS systems that did not have problems with setting VRD to voltages slightly past the maximum, will now display an error message saying "*Error: voltage in vrd() function is out of range*". There are two solutions for this problem: replace the AD562 DAC with a hand-picked one that has a gain error that will allow it to get to 10V (in span 10,UPO mode, for example) or

modify the test program so that it does not try to set vrd to 10V, but rather 10V-1LSB.

Output Wraps-around When Using *lts_printf()* Function

The *lts_printf()* function provides the user with a "printf" function for the PCLTS main test screen. It is especially useful for printing messages to the screen when developing or debugging a test program. It is very probable that the user may want to print the test number for the test from where messages are being printed, and will therefore want to include the *tnum* variable in the printed string:

```
lts_printf("\nThe Ref DAC at test number %ld is at %3.1f",tnum,rdvo);
```

The output for the above function call would look like this:

```
The Ref DAC at test number 3 is at 4.2
```

It is important that the proper type is used in the formatting string for the variables in the *lts_printf()* function. If *tnum* is formatted as simply int (%d) instead of type long int (%ld), as it is defined, the output may look like this:

```
The Ref DAC at test number 3 is at 4.2142987345871200000000002322
0202000000000000000000000000000012124000000000000000000000000000
0001500
```

The types of all global system variables can be found in the 2020LANG.H "include" file.

"Fail Bin 0" and "*Pass Bin 0"* Problems

When the classification system variables have not been properly set in a test program, parts may result in passing or failing *bin 0*.

The following conditions may result in this aberration:

- The *aclass* function has been called with ALL bins (*aclass(1,32)*), and a test has alarmed.
- The *pclass* function has been called with ALL bins (*pclass(1,32)*), and the part has passed.
- The *pclass* function has not been called at all in the program.

Appendix A

LTS Basic Cross-Reference Table

The following table is a cross-reference between the original LTS Basic language functions and statements and their equivalents in PCLTS C language. Please note that the definition of a function is reversed when you go from LTS Basic to PCLTS C. In Basic, *functions* are used to retrieve (use) the value of a variable and *statements* are used to set the value of a variable. In PCLTS C, *variables* are used to retrieve (use) the value and *functions* are used to set the value. Note that the following table indicates by each PCLTS variable if such variable should only be used in an expression or if it can also be set to a particular value.

LTS Basic	PCLTS C-Language
ABS function	**fabs()** function
ACLASS statement	**aclass()** function
ACLASV function	**aclasv** variable (use)
ACLASV statement	**aclasv** variable (set)
ADOSZ function	N/A
ADRDG function	**adrdg** variable (use only)
ADVIN function	**advin** variable (use only)
ALARM statement	**alarm** variable (use only)
ASC function	**atoi()** function (part of Microsoft C lib)
ATN function	**atan()** function (part of Microsoft C lib)
BASE statement	**base()** function
BAUD statement	N/A (use DOS *mode* command)
BIT function and statement	Use C language bit-wise operators
BSCT statement	Use IEEE-488 board library functions
BUSDAT statement	Use IEEE-488 board library functions
BUSIN statement	Use IEEE-488 board library functions
BUSRST statement	Use IEEE-488 board library functions
CALFAC function	Use PRINTCAL.EXE utility provided
CALFLG function	**sys_calibration_ocurred** variable (use only)
CALFLG statement	Not needed. System clears *sys_calibration_ocurred* variable after testing every device.
CALL statement	N/A
CHAN function	N/A
CLASS function	**class()** function
CLASV function	**clasv** variable (use)
CLASV statement	**clasv** variable (set)
CLEAR statement	N/A, use hdwclr() to reset hardware
CLOSE statement	N/A
CMV function	**cmv_value** variable (use only)
CMV statement	**cmv** function
CRB function	**crb_read()** function
CRB statement	**crb()** function
CREATC statement	N/A
CREATS statement	N/A
CRF function	**crf_read()** function

CRF statement	**crf()** function
DATA statement	N/A
DATE statement	Use Microsoft C time-related functions
DCAL function and statement	Use Engineering pull-down menu, option #1
DCB function	**dcb_value** variable (use only)
DCB statement	**dcb()** function
DCR function	**dcr()** function
DDB function	**ddb_value** variable (use only)
DDB statement	**ddb()** function
DDR function	**ddr()** function
DEBUG statement	N/A
DELETE statement	N/A
DFORM statement	N/A
DGATE statement	**dgate()** function
DIFF statement	**diff()** function
DIFFLG function	**difflg** variable
DIM statement	N/A
DINIT statement	N/A
DISK statement	N/A
DREAD statement	N/A
DWRITE statement	N/A
DSAVE statement	N/A
DUMP statement	N/A
DVDT statement	**dvdt()** function
ELSE statement	**else** statement
END statement	N/A
EOIFLG function	Use IEEE-488 board library functions
ERROR statement	N/A
ESCAPE statement	N/A
NOESC statement	N/A
EXP function	**pow()** function (part of Microsoft C lib)
FAIL function	Use **device_failed** and **test_downgraded** variables
FAIL statement	N/A
FCAL statement	Use F2 key once test program is loaded
FCLASS statement	**fclass()** function
FCLASV function	**fclasv** variable (use)
FCLASV statement	**fclasv** variable (set)
FDEL statement	**remove()** function (part of Microsoft C lib)
FLIST statement	N/A
FMOV and FMOVX statements	**fseek()** function (part of Microsoft C lib)
FOLVIN function	**folvin** variable (use only)
FOR/NEXT statement	**for** statement
FREAD statement	**fscanf()** function (part of Microsoft C lib)
FWRITE and FWRITEX statements	**fprintf()** function (part of Microsoft C lib)
GATE function	**SA_is_on**,...**SR_is_on** variables (use only)
GBGAIN function	**gbgain** variable (use only)
GBVIN function	**gbvin** variable (use only)
GET and PUT statements	N/A
GON and GOF statements	**gon()** and **gof()** functions
GOSUB statement	N/A
GOTO statement	**goto** statement
GSLINE function	N/A
HANDLR statement	**handlr()** function
HDWCLR statement	**hdwclr()** function
HIFLG function	**hiflg** variable (use only)

HLIMIT function	Use PRINTCAL.EXE utility provided
IF/THEN statement	**if** statement
INP function	**floor()**, **ceil()**, **fmod()** (part of Microsoft C lib)
INPUT statement	N/A
LEN function	**strlen()** function (part of Microsoft C lib)
LET statement	N/A
LIST command	N/A
LLIMIT function	Use PRINTCAL.EXE utility provided
LOG function	**log()** function (part of Microsoft C lib)
LSYNC function	**lsync** variable (use only)
LTRIG statement	**ltrig()** function
MAX function	**max_value** variable (use only)
MAX statement	**maxx()** function
MAXD function	**maxd_value** variable (use only)
MAXDstatement	**maxd()** function
MAXGBV function	**maxgbv** variable (use only)
MCH function	**strncmp()** function (part of Microsoft C lib)
MEAS statement	**meas()** function
MEM function and statement	N/A (use pointers in C)
MFBUF function	N/A
MSD function and statement	N/A
MWD function and statement	N/A (use pointers in C language)
MYAD statement	Use IEEE-488 board library functions
NEW statement	N/A
NKY function	N/A
NLOAD statement	N/A
NOISE function	**noise_value** variable (use only)
NOISE statement	**noise()** function
NULERR function	**nulerr** variable (use only)
NULFLG function	**nulflg** variable (use only)
NULL statement	**null()** function
NUMAVG function	**numavg** variable (use only)
ON statement	N/A
OPEN statement	**fopen()** function (part of Microsoft C lib)
PCLASS statement	**pclass()** function
PCLASV function	**pclasv** variable (use)
PCLASV statement	**pclasv** variable (set)
PIN and NIN functions	**pin** and **nin** variables (use only)
PLOAD statement	N/A
POP statement	N/A
PRINT statement	**lts_printf()** function
PSAVE statement	N/A
PULSE statement	**pulse()** function
RANDOM statement	**rand()** function (part of Microsoft C lib)
RDCODE function	**rdcode_value** variable (use only)
RDCODE statement	**rdcode()** function
RDCODE function	**rdcode_value** variable (use only)
READ statement	N/A
REM statement	Use "/* */" in C language programs
RES function	**res** variable (use only)
RESTOR statement	N/A
RESTRB statement	Use *meas()* function from Codeview command window
RETURN statement	**return** statement
REWIND statement	N/A
RTSTFL function	**rtstfl** variable (use only)

RUN command	Use F1 key once test program is loaded
SAVO, SBVO, SCVO, SDVO fncts	**savo**, **sbvo**, **scvo**, **sdvo** variables (use only)
SELECT statement	**select()** function
SERROR statement	N/A
SIN, COS functions	**cos()**, **sin()** functions (part of Microsoft C lib)
SIZE command	N/A
SNUM function	**snum** variable (use)
SNUM statement	**snum** variable (set)
SPAN statement	**span()** function
SPRINT statement	N/A
SPSTEP statement	N/A
SQR function	**sqrt()** function (part of Microsoft C lib
SRCODE function	**srcode_value** variable (use only)
SRCODE statement	**srcode()** function
SRH function	**strstr()** function (part of Microsoft C lib
SRQFLG function	Use IEEE-488 board library functions
SRVO function	**srvo** variable (use only)
STAT function	Use **initial_run** global variable
STAT statement	**initial_run** is reset by system after 1st device
STOP statement	N/A
SWITCH function and statement	N/A
SYS function	N/A
SYSAVE statement	N/A
TAB function	N/A
TEN function and statement	Not implemented
TEST function	Use value returned by **test()** function
TEST statement	**test()** function
THVO function	**thvo** variable (use only)
TIC function	Use Microsoft C time-related functions
TIME statement	Use Microsoft C time-related functions
TNUM function	**tnum** variable (use)
TNUM statement	**tnum** variable (set)
TTRIG statement	Not implemented
UNIT statement	N/A
VAL function and statement	N/A (use pointers in C)
VRD function	**vrd_value** variable (use only)
VRD statement	**vrd()** function
VSA, VSB, VSC, VSD functions	**vsa()**, **vsb()**, **vsc()**, **vsd()** variables (use only)
VSA statement	**vsa_value** function
VSB statement	**vsb_value** function
VSC statement	**vsc_value** function
VSD statement	**vsd_value** function
VSR function	**vsr_value** variable (use only)
VSR statement	**vsr()** function
VTH statement	**vth()** function
VTH function	**vth_value** variable (use only)
WAIT statement	**lts_wait()** function

Appendix B

STDF Datalog File Format

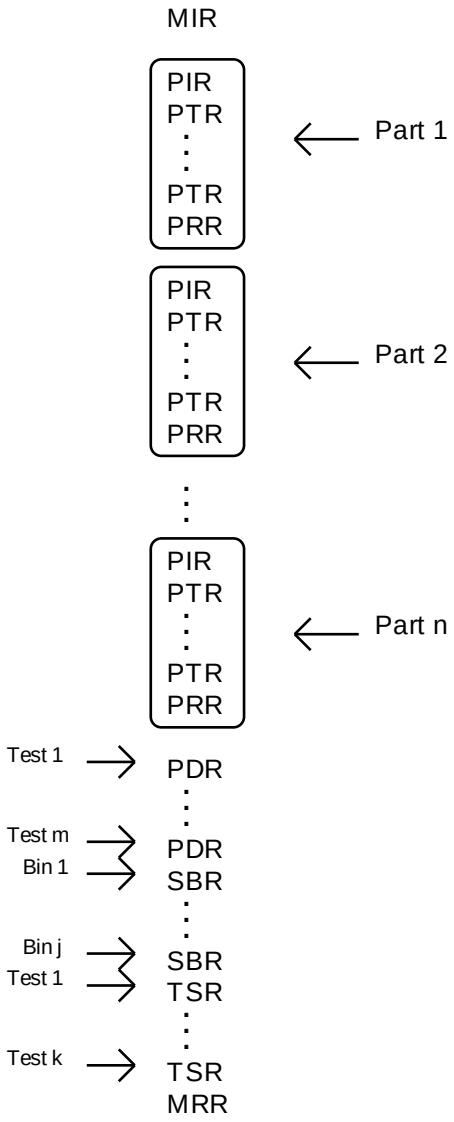
The PCLTS system generates binary datalog files using the STDF data format. STDF (**S**tandard **T**est **D**ata **F**ormat), developed by Teradyne, has quickly become a widely accepted industry standard.

PCLTS datalog files adhere to Version 3 of the standard. For a complete specification please contact Teradyne Inc. in Boston, Massachusetts.

STDF files consist of a series of records and fields. Some records and fields are required and some are optional.

STDF File Record Ordering

The SDTF standard allows for several different ordering options of the data records. The following is the one used by the PCLTS system:



STDF Record Ordering

Fields Used Within Each Record

The following is a listing of the records, and fields within each record, used by PCLTS datalog files.

Master Information Record (MIR)

REC_LEN
REC_TYPE
REC_SUB

CPU_TYPE (Type 2 is used)
STDF_VER (Ver 3 is used)
MODE_COD
STAT_NUM
TEST_COD
START_T
LOT_ID
PART_TYP
JOB_NAM
OPER_NAM
NODE_NAM
TSTR_TYP

Part Information Record (PIR)

REC_LEN
REC_TYP
REC_SUB

HEAD_NUM (always set to 1)
SITE_NUM (always set to 1)
X_CORD (always set to -32768)
Y_CORD (always set to -32768)
PART_ID

Parametric Test Result Record (PTR)

REC_LEN
REC_TYP
REC_SUB

TEST_NUM
HEAD_NUM (always set to 1)
SITE_NUM (always set to 1)
TEST_FLG
PARM_FLG
RESULT

Parametric Test Description Record (PDR)

REC_LEN
REC_TYP
REC_SUB

TEST_NUM
DESC_FLG (always set to 0)
OPT_FLAG (always set to 0)
RES_SCAL
UNITS
RES_LDIG
RES_RDIG
LLM_SCAL
HLM_SCAL
LLM_LDIG
LLM_RDIG
HLM_LDIG
HLM_RDIG
LO_LIMIT
HI_LIMIT

Software Bin Record (SBR)

REC_LEN
REC_TYP
REC_SUB

SBIN_NUM
SBIN_CNT
SBIN_NAM

Test Synopsis Record (TSR)

REC_LEN
REC_TYP
REC_SUB

TEST_NUM
EXEC_CNT
FAIL_CNT
ALRM_CNT
OPT_FLAG (always set to 3FH)

Master Results Record (MRR)

REC_LEN
REC_TYP
REC_SUB

FINISH_T
PART_CNT
RTST_CNT
ABRT_CNT (always set to -1)
GOOD_CNT
FUNCT_CNT (always set to -1)
DISP_COD (always set to -1)
USR_DESC

Appendix C

Interconnecting Cables

LTS PORT1 and PORT2 Ribbon Cables

Faulty Ribbon Cables

The two 25-pin connectors on the back of the LTS tester are connected to two 26-lead ribbon cables which end in 26-pin IDC connectors inside (these two cables should have been labeled "A" and "B" in step 7 of the section labeled "Installing the Bus Driver Board" in this manual). Several of these cables have been found to have unreliable connections where the DB-25 connector crimps the ribbon cable. These faulty contacts may have gone unnoticed when the LTS was operated in its original mode because in the serial port not all lines are used and because the noise margin of the RS-232 standard is much higher (due to the use of 12-volt signals). With this extra noise margin, contacts with several tens of ohms could still operate properly.

In the PCLTS system, however the two DB-25 connectors are not used as serial ports, but instead, to bring the CPU emulator board signals to the LTS bus. It is therefore crucial that all lines have continuity since all of them are used. Also, signals are 5-volt TTL, which have significantly less noise margin than 12-volt RS-232.

Measuring Cable Resistance

There may be two type of problems with the ribbon cable assemblies: high resistance or simply no electrical continuity at all. To insure proper electrical integrity, measure the resistance of each one of the lines on both ribbon cables and make sure they don't exceed 1 or 2 ohms (it should typically be around 0.3 ohms). To perform the check, connect the leads of an ohm-meter between the DB-25 connector pins at the back of the LTS and the 26-pin IDC connectors at the other end of the ribbon cable. Insert a small piece of wire into the IDC connector to reach the connector's individual pin sockets. Do this for every pin on the connector. Table C-1 below shows the pin correspondence between the two types of connectors.

Important Note:

The DB-25 connectors used have an internal mapping that is non-standard. Generally when a 26-lead ribbon cable is connected to a 25-pin DB-25 connector, it's the outermost lead of the ribbon cable that is left disconnected. In the connectors used by the LTS, however, it's the **middle** lead of the ribbon cable that is not connected.

LTS Ribbon Cables Mapping

These are the two 26-lead ribbon cables that connect the two connectors labeled PORT1 and PORT2 on the back of the LTS tester with IDC-26 connectors. These cables should have been labeled **A** and **B** in step 7 of the *Installing the Bus Driver Board* section at the beginning of this manual. These cables use a non-standard DB-25 connector which has an unusual mapping, as illustrated by the following table and drawing:

Table C-1: Ribbon Cables Inside LTS Tester

DB-25 Pin #	IDC-26 Pin #
1	1
2	2
3	3
4	4
5	5
6	6
7	20
8	21
9	22
10	23
11	24
12	25
13	26
14	14
15	15
16	16
17	17
18	18
19	19
20	8
21	9
22	10
23	11
24	12
25	13
—	7

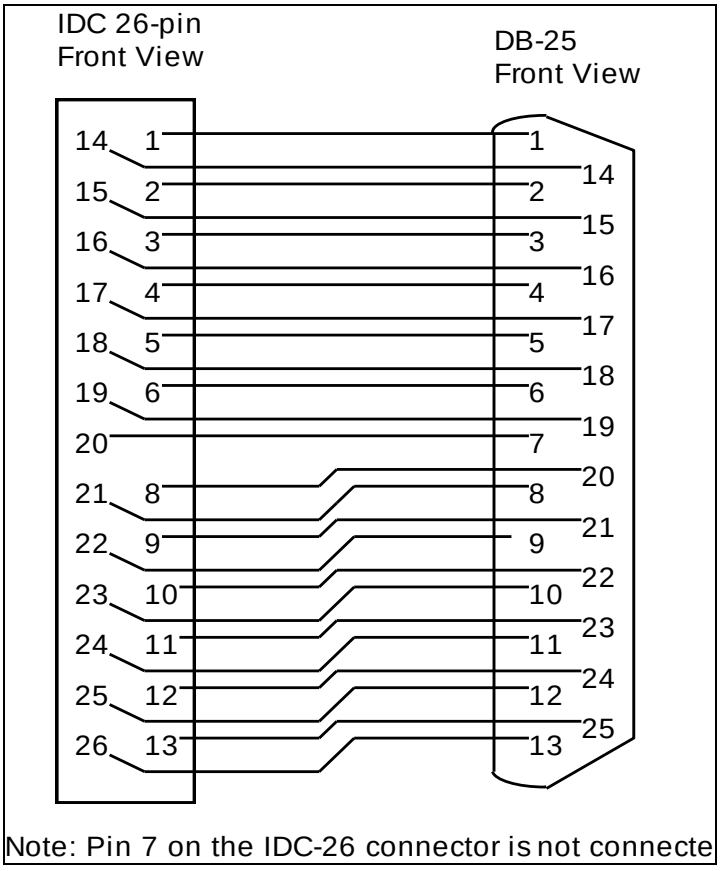


Figure C-1; Pinout for ribbon cables inside LTS tester

Emulator Extender Bracket Cable

This is the cable that connects J2 on the CPU Emulator board to the auxiliary bracket inside the PC. This is a standard pin-to-pin cable assembly, as shown on table C-2 below:

Table C-2: Emulator Auxiliary Bracket Ribbon Cable (Inside PC)

DB-25 Pin #	IDC-26 Pin #
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17
18	18
19	19
20	20
21	21
22	22
23	23
24	24
25	25
—	26

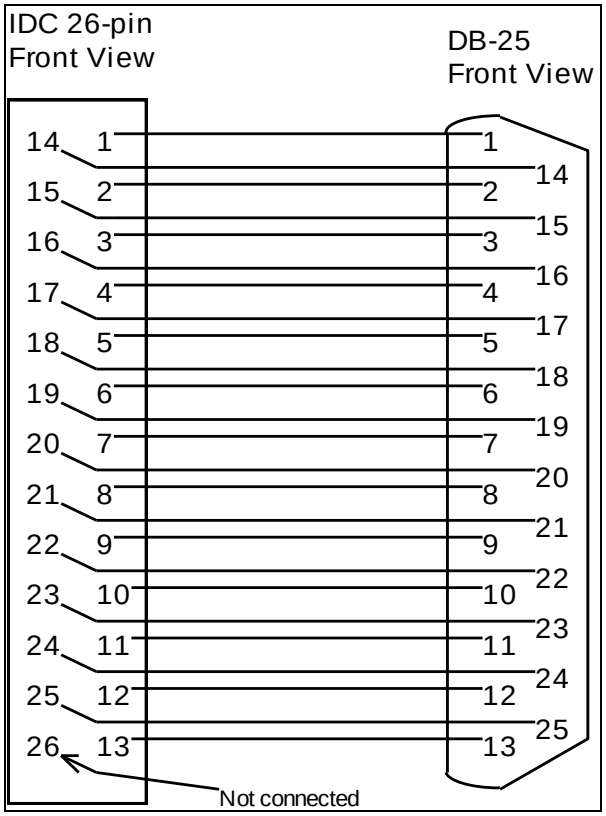


Figure B-2; Emulator Auxiliary Bracket Pinout

Cables Connecting PC to LTS Tester

These are the two 25-lead long cables that link the LTS tester with the PC. One of these cables connects the connector labeled PORT1 on the back of the tester with the connector on the CPU Emulator board inside the PC. The other connects the connector labeled PORT2 on the back of the tester with the auxiliary bracket on the PC.

Both cables are identical and consist of standard shielded pin-to-pin (straight-through) DB-25 male to DB-25 male cables. These are sometimes referred to as "serial port cables" and are readily available.

Important note: If replacing these cables, make sure the new cables are truly straight-through. "Null modem" cables should not be used, since they are not fully straight-through.

Appendix D

Gain Settings

When the *maxx()* or *maxd()* functions are called, a system variable is set which determines what gain will be set when the next *meas()* or *diff()* function is called. Before taking a measurement the system will set the gain relays on the measurement board.

The programmable gain stage of the measurement system consists of relays that connect parallel combinations of resistors to form the feedback element in a non-inverting op-amp. There is a finite number of resistors and therefore a finite number of possible gain settings. The range of possible gain values spans from 0.8 to 1077.6 and eventhough there is a total of 64 possible values, only 11 make sense being used. This is because gain values are grouped in 11 widely separated clusters.

Due to differences in circuit topologies, not all calls to *maxx()* and *maxd()* result in the same gain settings. In fact, there are four different groups of measurements.

***select(SA,CUR)*-Type Measurements**

SA,CUR
SB,CUR
SC,CUR
SD,CUR

***select(SR,CUR)*-Type Measurements**

SR,CUR
TH,CUR

***select(SA,VLTG)*-Type Measurements**

SA,VLTG
SB,VLTG
SC,VLTG
SD,VLTG

***select(LA,LC)*-Type Measurements**

LA,LC	LB,LC
LA,LD	LB,LD
LA,TN	LB,TN
LA.GD	LB,GD
SR,VLTG	TH,VLTG
GD,LC	RD,LC

GD,LD
GD,TN
GD,GD
GD,RD
TN,LC
TN,LD
TN,RD
RD,LD
RD,TN
RD,GD
RD,RD
TN,TN
TN,GD

The following tables show the possible gain settings and the *maxx()* and *maxd()* calls that result in them, for each type of measurement.

Table D-1
select(SA,CUR)-Type Measurement

maxx()* or *maxd()	**Resulting Gain**
0.00000 to 0.00178	-1077.6
0.00178 to 0.00746	-257.6
0.00746 to 0.00956	-200.8
0.00956 to 0.03342	-57.6
0.03342 to 0.04700	-40.8
0.04700 to 0.10921	-17.6
0.10921 to 0.15001	-12.8
0.15001 to 0.34280	-5.6
0.34280 to 0.48000	-4.0
0.48000 to 0.80001	-2.4
0.80001 to 10.0000	-0.8

Table D-2
select(SR,CUR)-Type Measurement

maxx()* or *maxd()	**Resulting Gain**
0.00000 to 0.00010	-1077.6
0.00010 to 0.00038	-257.6
0.00038 to 0.00048	-200.8
0.00048 to 0.00168	-57.6
0.00168 to 0.00236	-40.8
0.00236 to 0.00546	-17.6
0.00546 to 0.00750	-12.8
0.00750 to 0.01714	-5.6
0.01714 to 0.02402	-4.0
0.02402 to 0.04002	-2.4
0.04002 to 10.0000	-0.8

Table D-3

select(SA,VLTG)-Type Measurement

maxx()* or *maxd()	**Resulting Gain**
0.00000 to 0.01782	-1077.6
0.01782 to 0.07462	-257.6
0.07462 to 0.09561	-200.8
0.09561 to 0.33400	-57.6
0.33400 to 0.47000	-40.8
0.47000 to 1.09200	-17.6
1.09200 to 1.50002	-12.8
1.50002 to 3.42801	-5.6
3.42801 to 4.80001	-4.0
4.80001 to 8.00001	-2.4
8.00001 to 10.0000	-0.8

Table D-4
select(LA,LC)-Type Measurement

maxx()* or *maxd()	**Resulting Gain**
0.00000 to 0.00890	-1077.6
0.00890 to 0.03732	-257.6
0.03732 to 0.04780	-200.8
0.04780 to 0.16700	-57.6
0.16700 to 0.23501	-40.8
0.23501 to 0.54600	-17.6
0.54600 to 0.75000	-12.8
0.75000 to 1.71401	-5.6
1.71401 to 2.40000	-4.0
2.40000 to 4.00001	-2.4
4.00001 to 10.0000	-0.8

Appendix E

Specifications

CPU Emulator Board (TSA-2500)

PC I/O bus addresses used	: 310H-31FH
PC Hardware interrupts used	: none
On-board timer resolution	: 1us
Type of PC slot required	: one 16-bit (and an adjacent 8 or 16-bit)
Height	: 3.7 inches
Length	: 13.3 inches

PC Requirements

Type	: IBM-compatible PC (not included)
I/O bus	: ISA bus
CPU	: from 486-25MHz to Pentium 166MHz
Video	: VGA color graphics
RAM	: 8 Mbytes or more
Hard Disk	: 500 Mbytes or more recommended
Mouse	: Microsoft-compatible mouse
Case	: Standard mini-tower or comparably-sized desktop. The CPU emulator board is a full-length and full height board.
Networking (optional)	: Any networking system that supports a DOS virtual drive on the server is appropriate. We recommend using an Ethernet board with PCNFS or Novell software.
GPIB interface board	: National Instruments GPIB PC-II interface card.

Software Requirements

Operating System	: MS DOS version 6.0 or later
C Compiler	: Microsoft Visual C++ version 1.0 or higher.
Available DOS Memory	: At least 610 kBytes. Use Memmaker or a third-party utility to load TSRs and device drivers in "high" memory.

Index

-2-

2020LANG.H 4-1

-3-

386MAX memory manager 3-7, 13-2

-A-

aclass() function 10-1
aclasv variable 11-2
adrdg variable 11-3
advin variable 11-3
alarm variable 11-1
array sizes in C 5-9
array subscripts 8-9
auto_test_parameters_and_bin() function 10-1
auxiliary register 12-1

-B-

base() function 10-2
base_value variable 11-1
bin_current_test() 4-2
bin_current_test() function 10-2
bin_descriptions[ ] variable 11-4
binning 4-3
board diagnostics 6-6, 12-1
BRDINSTL.EXE utility 9-8
breakpoints 5-3
by-passing calibration 6-20

-C-

cable resistance 16-1
cable, auxiliary bracket 16-4
cables, LTS console to PC 16-6
cables, PORT1 and PORT2 inside LTS console 16-2
calibration factors 9-6
check_fail() function 4-2, 10-3
clas() function 10-3
classification system 4-3
clasv variable 11-4
clearing the summary data 6-16
cmv() function 10-4
cmv_value variable 11-5
Codeview debugger 5-1
Codeview hangs up 13-1

collecting data points 6-21
compiler switches 4-15
compiling the test program 4-15
conversion examples 8-5
CONVERT.EXE utility 8-1
CONVERT.RPT file 8-7
correlation criteria file 9-1
CORRLGEN.EXE utility 9-2
crb() function 10-4
crb_read() function 10-4
creating a test program 4-1
crf() function 10-5
crf_read() function 10-5
Ctrl-Break to abort 6-24
current setup window 6-23
CURRENT.STS file 5-7, 13-1

-D-

data types in C 5-8
data-logging 6-12, 6-23
datalog file path 6-20
dcb() function 10-6
dcb_value variable 11-5
dcr() function 10-6
ddb() function 10-7
ddb_value variable 11-6
ddr() function 10-7
debugging the test program 5-1
define_bin_descriptions() function 4-5
definitions 4-6
development menu 6-7
device_failed variable 11-6
dgate() function 10-7
DIAGNOSE.EXE utility 12-1
diagnostics program 2-4
diff() function 10-8
difflg variable 11-7
directories tree for PCLTS 3-4
displaying test limits 6-15
disqualify_bin() function 4-4, 10-8
DOS memory 3-7
down-grading example 4-4
downgrades 4-3
dvdt() function 10-9

-E-

EXCEL 6-5
executing functions while debugging 5-5
exiting the shell 6-6

-F-

fail bin 4-3
family board ID 6-19
fast_diff() function 10-9
fast_meas() function 10-10
fclass() function 10-11
fclasv variable 11-7
file transfer (LTS Basic to PC) 8-1
folvin variable 11-8
forcing calibration 6-22
forcing through failures 6-17
function keys 6-22
function prototypes 4-7

-G-

gain settings 17-1
gbgain variable 11-8
gof() function 10-11
gon() function 10-11
GOSUB 32000 subroutine 4-2
goto, avoid using 7-1
gotos 8-10

-H-

handler configuration file 6-3
handler configuration files creation 6-7
handler port disabling 6-19
handlr() function 10-12
hardware installation 2-1
hdwclr() function 10-12
HDWCLR.EXE utility 9-8
hiflg variable 11-9

-I-

include files 4-6
initial_run variable 11-9
INSTALL program 3-4
installing the Bus Driver Board 2-3
installing the CPU Emulator Board 2-1
installing the PCLTS system software 3-4
interconnecting cables 16-1

-L-

large memory model 4-15
library functions 10-1
limit file format 4-8
limit table 4-7
limits_type[ ] variable 11-10
line synch 12-3
linker switches 4-16
linking 4-16
loading "high" 13-2

loading the limit table 4-8
loading the test program 6-3
lsync variable 11-10
ltrig() function 10-13
LTS Basic cross-reference table 14-1
lts_printf() function 10-13
lts_wait() function 10-14
lts_wait_secs() function 10-14
LTSSHEL system shell 6-1
LTSSHEL.INI file 3-5

-M-

main() function 4-1
max_value variable 11-12
maxd() function 10-15
maxd_value variable 11-11
maxgbv variable 11-11
maxx() function 10-14
meas() function 10-16
message_to_user() function 10-16
modifying variables 5-6
msd() function 10-17
msd() not needed 7-3

-N-

nin variable 11-12
noise() function 10-17
noise_value variable 11-13
not enough available memory 13-1
nulerr variable 11-13
nulflg variable 11-14
null() function 10-18
numavg variable 11-14

-O-

optimizing the program 7-1
overview 1-1

-P-

pclass() function 10-19
pin variable 11-15
PRINTCAL.EXE utility 9-4
printer setup 2-4
printing a summary sheet 6-16
privilege levels 6-1
processing a datalog file 6-4
product_name[ ] variable 11-15
program_run_by_Verify_Setup variable 11-16
pulse() function 10-19

-Q-

QEMM memory manager 3-8, 13-2

-R-

rdcode() function 10-20
rdcode_value variable 11-17
rdvo variable 11-17
re-directing the printer port 2-5
re-testing devices 6-18
read_family_board_id() function 10-20
record_data_point() function 6-21, 10-21
reducing test times 7-2
reference file 9-3
res variable 11-16
rtstfl variable 11-18
run_time_error_msg_to_user() function 10-21
running Codeview 6-7
running verify setup 6-4, 9-1

-S-

SA_is_on variable 11-19
sample datalog 6-15
sample test program 4-11
savo variable 11-18
SB_is_on variable 11-20
sbvo variable 11-19
SC_is_on variable 11-21
screen flicker 5-6
scvo variable 11-20
SD_is_on variable 11-22
sdvo variable 11-21
select() function 10-22
selecting the test program 6-2
serial numbers 6-18
serial port ribbon cables inside LTS 12-3
serial printers 2-5
set_gain_relays() function 10-22
single-stepping 5-3
snum variable 11-22
software installation 3-1
source profiler utility 7-3
span() function 10-23
span_value variable 11-23
specifications 18-1
SR_is_on variable 11-24
srcode() function 10-23
srcode_value variable 11-23
stack overflow 4-17
starting a new lot 6-17, 6-23
STDF data file format 15-1
STDF records ordering 15-2

structured programming 7-1
summary sheet 6-16
sys_calibration_ocurred variable 11-24
sys_reset() function 10-24
system configuration editor 6-10
system requirements 2-1
system reset 6-21

-T-

test program creation 4-1
test time displaying 6-18
test() function 10-24
test_conditions_code[ ] variable 11-25
test_downgraded variable 4-4, 11-25
test_mode_code[ ] variable 11-26
test_parameters() function 10-25
test_program() function 4-5
thvo variable 11-26
timer 12-2
tnum 4-2, 8-8
tnum variable 11-27
tone option 6-18
trapping 5-3
trial files 9-2
troubleshooting system software 13-1

-U-

user switches 6-18
user_after_testing_part() function 4-6
user_before_testing_part() function 4-6
user_switch_n variables 11-27
using a limit table 4-7

-V-

variable declarations 4-7
variable names 7-1
variable types 8-9
vrd() function 10-27
vrd_value variable 11-28
vsa() function 10-25
vsa_value variable 11-28
vsb() function 10-25
vsb_value variable 11-29
vsc() function 10-26
vsc_value variable 11-29
vsd() function 10-26
vsd_value variable 11-30
vsr() function 10-27
vsr_value variable 11-30
vth() function 10-28
vth_value variable 11-31

-W-

watching variables 5-4