TERADYNE Z1800-Series

# Digital Function Processor™ User's Guide

Teradyne, Inc.
Assembly Test/Walnut Creek
2625 Shadelands Drive, Walnut Creek, California 94598-2597
Publication Number 1800M046 Copyright Teradyne Inc.

# Digital Function Processor User's Guide

## Manual History

- Fifth Edition, May 1998, Version B.1 software
- Fourth Edition, October 1995, Version B.0 software
- Third Edition, September 1994, Version A.0 software
  Changes the manual's name from *PROMPTest II User's Guide* to *Digital Function Processor User's Guide*
- Second Edition, July 1994, Version A.0 software
- First Edition, June 1994, Version A.0 software
- Preliminary Edition, December 1993

Publications No. 1800M046

# Contents

## Illustrations

## Tables

# Standard Flash Memory Applications 1

## Introduction

The Digital Function Processor (DFP) is a flexible platform for implementing value-added tests and product functional tests on the Z1800-Series board testers.

Initial DFP applications program nonvolatile memories such as EEPROM and Flash ROM. Future applications may include full cell memory testing and product functional testing. This manual concentrates on the nonvolatile memory programming applications.

Target devices for current applications include nonvolatile memories, electrically erasable PROMs, in-system programmable logic devices, and microcontrollers that can be serially bootstrapped.

DFP has three main modes of operation:

- free-air
- serial boot
- handshake

In the free-air mode, DFP writes user data files directly into DUT (device under test) nonvolatile memories. This mode is useful when the DUT's nonvolatile memory is fully accessible.

In the serial boot mode, DFP loads a program into DUT random access memory (RAM) using a serial bootstrap protocol, the DUT program being designed to write into the internal EEPROM of the DUT CPU.

In the handshake mode, DFP bootstraps the DUT's CPU (central processing unit), then feeds parallel data to it on demand. DFP provides its own data path and does not depend on the VP (vector processor) option. The DUT CPU program handles the details of the nonvolatile memory interactions.

DFP is compatible with the Z1805, Z1808, Z1840, Z1850, Z1860, Z1866, Z1880, Z1884, and Z1890 testers. The Z1800 and Z1820 cannot accommodate the DFP, due to lack of rack mounting facilities for the chassis.

**Note: The factory can quote custom modifications and retrofits for existing testers, including the Z1800 and Z1820. However, retrofits may require adding the Relay Array Board or upgrading to the current Programmable Power Supply control board to provide a DFP bootstrap control relay.**

## Overview

### Software

Digital Function Processor software is completely separate from the main body of Z1800-series system software, so updates to either software can occur independently of the other.

The DFP software package includes examples of source code and library routines. You may use the C compiler as needed to develop programs that run in the DFP computer, to develop external programs that are called from in-circuit test pages, to develop Test Toolbox programs, and to develop occasional software tools such as format conversions. The C programs provided with DFP are written for MicroSoft QuickC version 2.5 and Turbo C++ 3.0 for DOS.

C programs for the main tester PC are normally developed and compiled on the main tester PC.  Those for the DFP computer are best compiled and debugged on the DFP computer. The main tester PC can stay in a test step in edit mode while you make changes and recompile the DFP code.

### Hardware

DFP hardware consists of

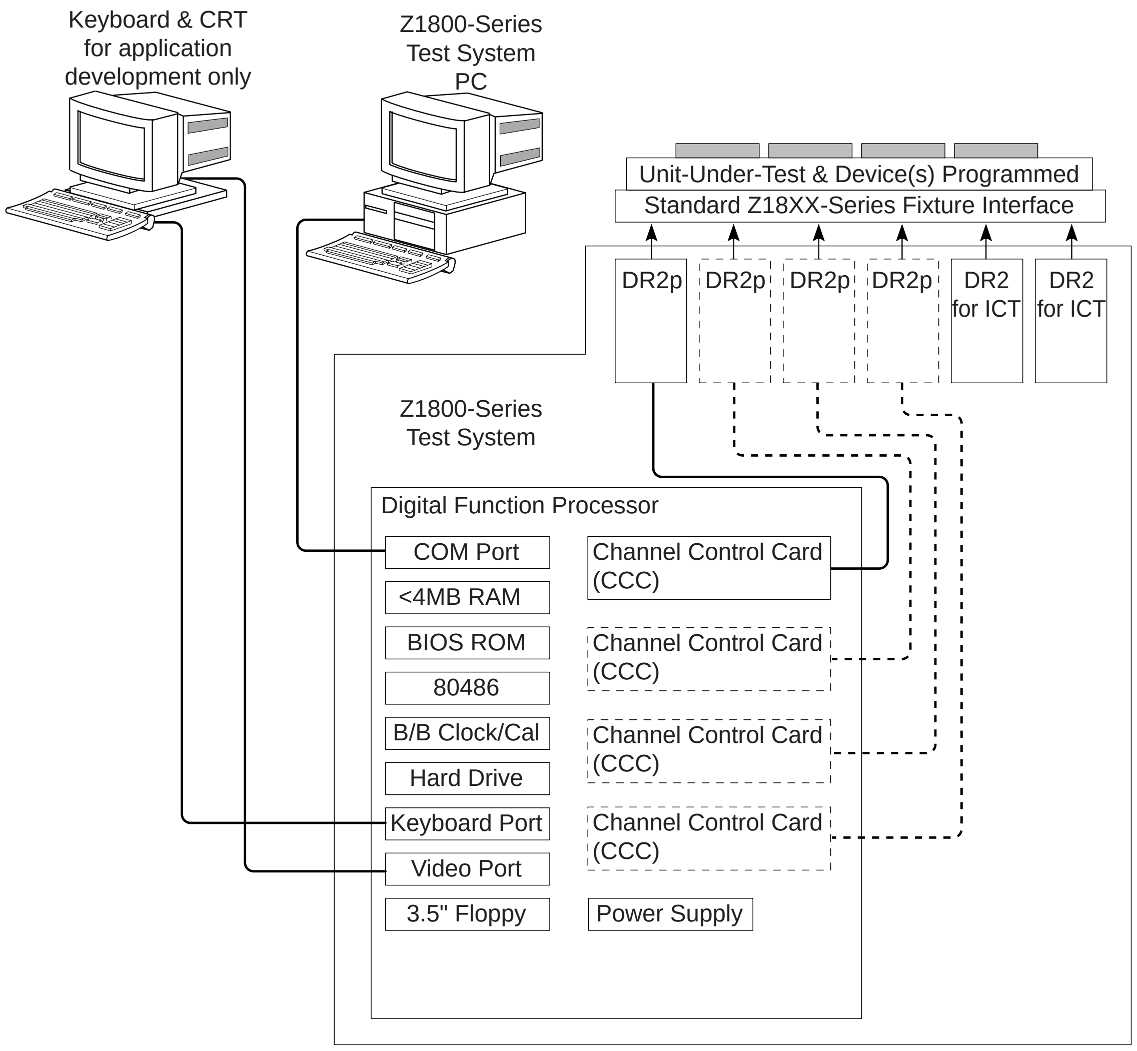
- a chassis
- a computer
- Teradyne Channel Control Cards (CCCs)
- a Driver Receiver 2p (DR2p) board in the Z1800-series tester

The DFP chassis contains a PC clone style computer. The DFP computer has its own hard disk with 4 mb of RAM, and runs MS-DOS operating system version 6.2 or later.  The motherboard of the DFP has conventional expansion-board slots which support Teradyne Channel Control Cards (CCCs). The DFP computer is connected to the Z1800-Series tester’s computer by a null-modem cable between the COM ports in each computer.

A DR2p board has all the in-circuit testing features of the standard 32-channel DR2. In addition, the DR2p channels have features that allow them to serve the needs of DFP. DR2p channels pass all standard Z1800-series self-tests and can be used as ordinary in-circuit test channels for digital Gray code, vector processors, analog, and mixed signal purposes in applications not using DFP functions. Specialization of certain DR2p channels to certain functions demands special wiring for fixtures in applications that use DFP functions.

DFP and the Z1800-series main computer do not operate as equals on a network.  Rather, DFP is a slave to the Z1800-series main computer. The Z1800-series main computer sends commands to the DFP computer. The command interface consists of Com ports connected by a null modem cable.  The bit rate in this cable is programmable.

Figure 1.1 Hardware Interconnect Diagram



Note: Digital Function Processor shown configured with
one Channel Card installed,
expandable up to four CCCs.

## DFP and DR2p Slot Populations

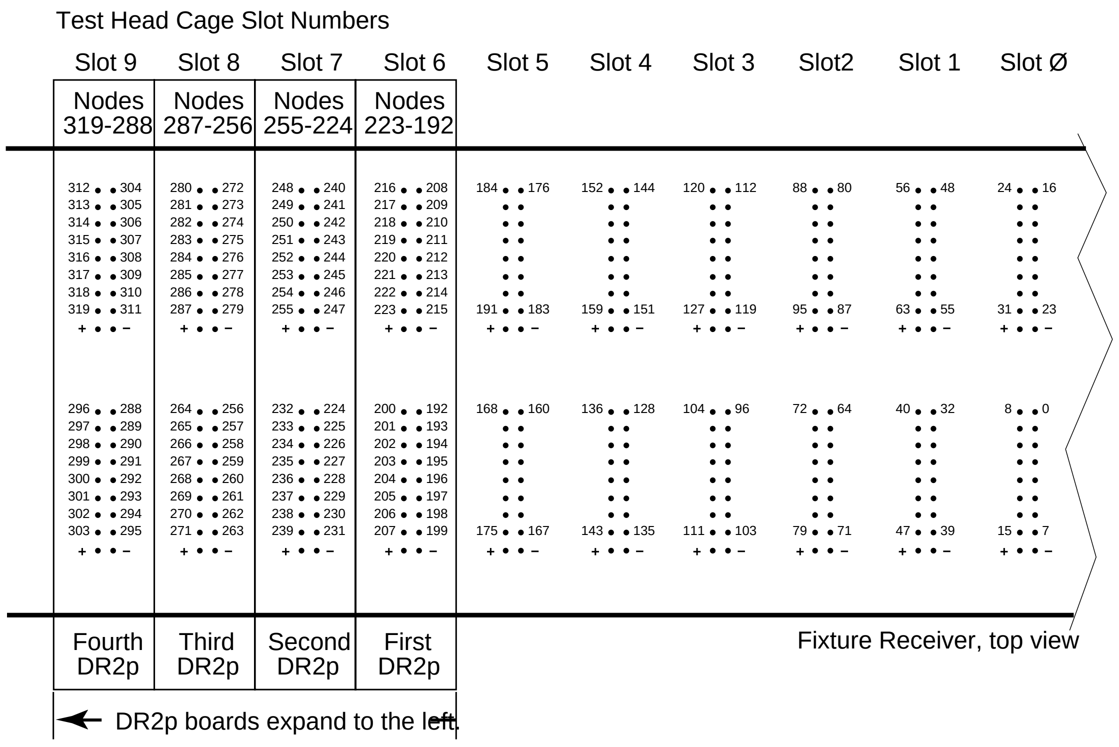
In simple serial boot applications, one board in serial mode is sufficient to serve two DUT microcontrollers. You can add a second board to allow service of a third and fourth microcontroller in cases where, for example, the board being tested has four daughter panels.

In serial-boot-with-handshake applications, one board in serial mode serially bootstraps up to two DUT microcontrollers. A second board in handshake mode can provide 8-bit wide handshake data to each of two DUT microcontrollers, or 16-bit wide handshake data to a single DUT microcontroller.

In flash-in-free-air applications, one board in address mode generates addresses for DUT memory. A second board in parallel or data mode provides 16-bit wide data and control signals to the DUT flash memories.

Teradyne has adopted a convention for the location of DR2p boards associated with the individual CCCs, although the DR2p boards can in fact be placed anywhere in the cage. For the sake of being usable with small (less than 320 node) fixtures, DFP channels are placed in the top four driver-receiver board locations of the part of the tester available to 320-node fixtures.

Figure 1.2 Preferred Locations for DR2p Boards in DFP-Equipped Testers



## DR2p Boards

DR2p boards have all the features of DR2 boards, including the ability to run all standard Z1800-series in-circuit tests and diagnostic tests on all channels. In addition, DR2p boards can program nonvolatile memory (NVM). Signals sent from DFP to the DUT pass through the in-circuit drive amplifiers on the DR2p boards. Thus they are capable of backdrive where the situation calls for it. No actual backdriving takes place if the DUT is able to place its internal buses in a high impedance state. DR2p boards provide parallel sensing facilities. DFP and its DR2p boards are compatible with VP or THC versions of Z1800-series testers.

Because DFP pin functions are varied, and in many cases different from, normal in-circuit functions, all pins are not equal.  Some of DFP pin functions are dedicated to specific groups of node numbers. Addresses appear at one group of pins in the fixture receiver, parallel data at another, serial ports at another, and special voltages at another.

Fixtures for plain in-circuit testing may be wired at random, without care as to where the DFP functions appear. However, fixtures used for nonvolatile memory applications rely on pre-dedicated wiring to connect each part of DFP to its specific target nodes in the DUT.

The inputs of the first 24 channel-driver amplifiers in the DR2p board can be driven either by the native DR2 digital stimulus generators, or by signals coming in over the cable from the CCC. DFP can use these 24 driver circuits directly as needed.

The first 24 channels of a DR2p provide parallel reception so that DFP can read back multiple bytes from the DUT. The readback facility provides a ready self-test mechanism for the first 24 channels.

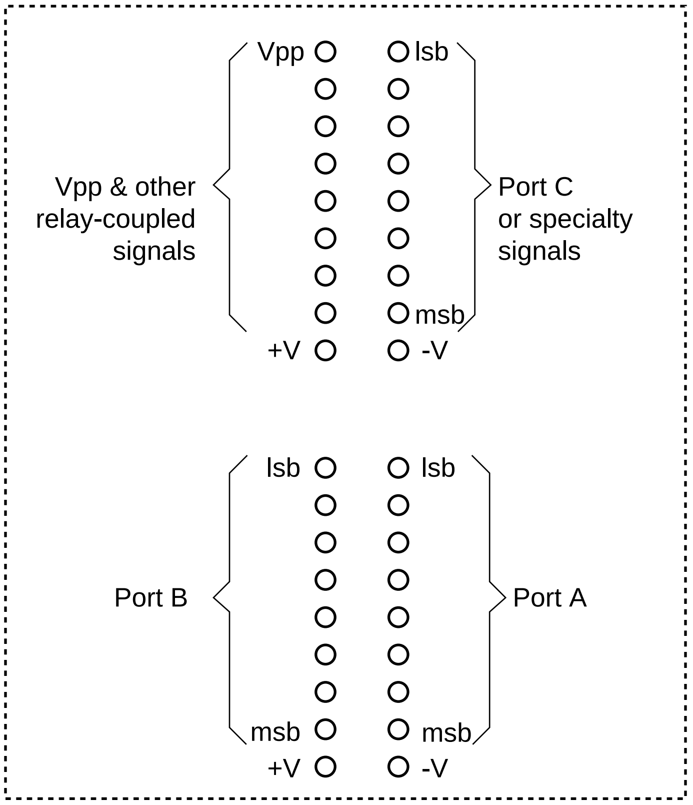
The first 24 channels also provide local enabling of two 8-bit banks by a DUT-generated strobe. This is used in the handshake mode to allow the DUT to view two byte-wide groups of DR2p channels as virtual input ports on the DUT data bus.

The eight high-order channels have relays which provide direct metallic conduction paths between CCC and DUT. These relays are individually controlled by the software in the DFP computer. They are used to force and/or sense the Vpp level (usually 12 volts) or other voltage levels that cannot pass though digital drive and sense circuits. Except for the Vpp and RS232 functions, these relays are reserved for future use or for custom CCC development.

The following illustrations show

- a generic pinout of the DR2p board
- an example of fixture node assignments for flash in free air operations
- an example of fixture node assignments for serial bootstrap and handshake operations

Figure 1.3 Generic Pinout of DR2p Board


Vpp
lsb
Vpp & other relay-coupled signals
Port C or specialty signals
msb
+V
-V
lsb
lsb
Port B
Port A
msb
msb
+V
-V

Generic pinout of DR2p Board.
Actual functions depend on mode selections.

# Software Architecture 2

## Introduction

As mentioned earlier, Digital Function Processor software is completely separate from the main body of Z1800-series system software, so updates to either software can occur independently of the other.

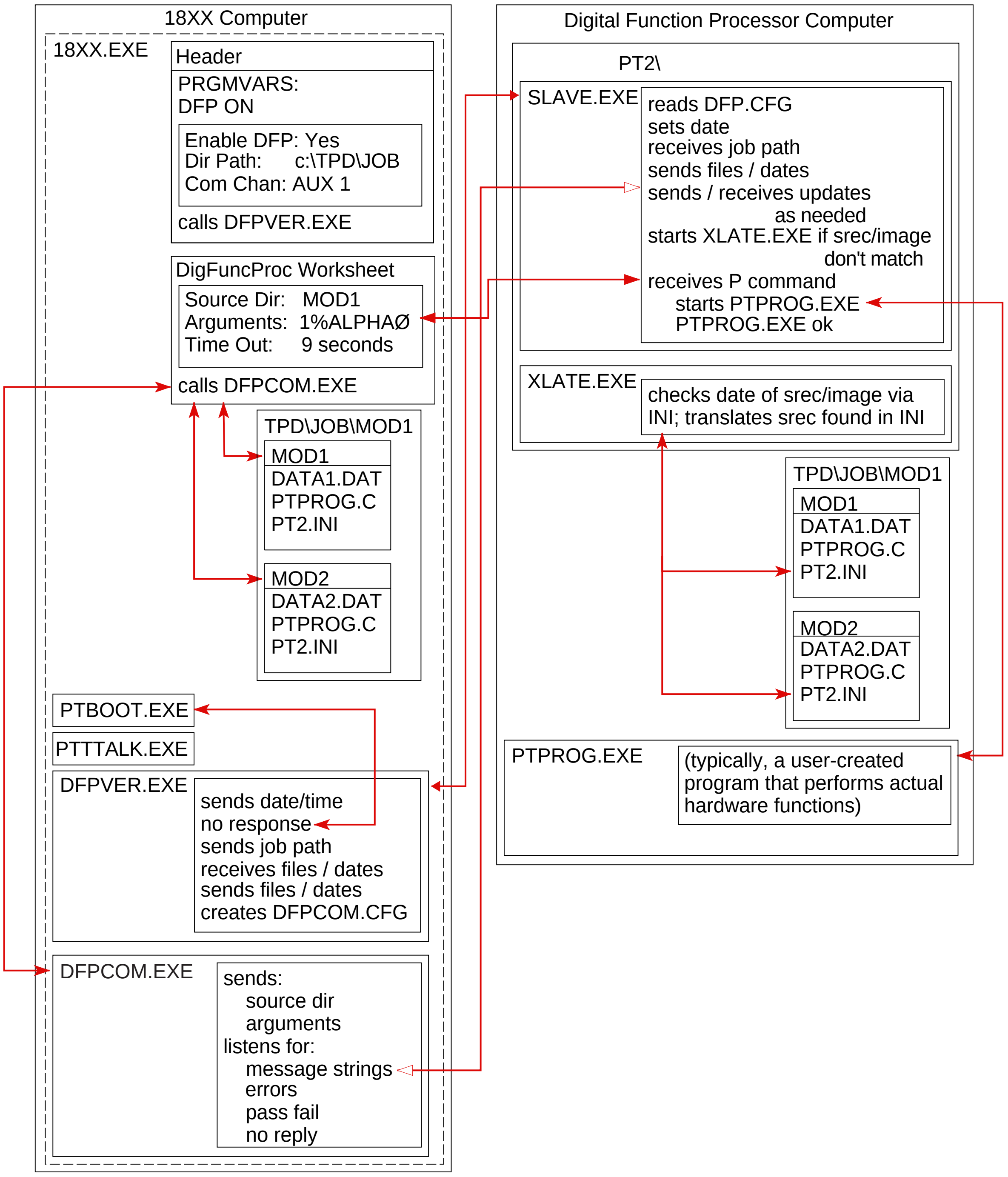
The DFP software package includes examples of source code and library routines. You may use the C compiler as needed to develop programs that run in the DFP computer, to develop external programs that are called from in-circuit test pages, to develop Test Toolbox programs, and to develop occasional software tools such as format conversions. The C programs provided with DFP are written for MicroSoft QuickC version 2.5 and Turbo C++ 3.0 for DOS.

C programs for the main tester PC are normally developed and compiled on the main tester PC. Those for the DFP computer are best compiled and debugged on the DFP computer. The main tester PC can stay in a test step in edit mode while you make changes and recompile the DFP code.

DFP is controlled by a tester PC COM port over a serial RS-232 link. The test program sends commands to DFP through the AUX PORT facility.

As illustrated, one general software system is in place during board testing for any given custom or flash-in-free-air application. (The values and choices shown for the PRGMVARS and DigFuncProc worksheet fields are examples only; they may vary during actual testing.)

Figure 2.1  Software System Used During Board Testing



## User Interaction

Users interact in three primary ways with DFP. In order of increasing interaction intensity, these are

- as operators testing boards
- as technicians verifying or troubleshooting DFP
- as C programmers developing custom applications

### Operators

An operator testing a board will generally be unaware of the presence of DFP.

### Technicians

For verifying and troubleshooting DFP, a test program is provided as a Z1800-series software subdirectory. Its location is TPD\PTDIAGS\ICT.TST. In a corresponding subdirectory of DFP's disk is a DFP program, PTPROG.EXE, that responds to a variety of commands aimed at verifying and troubleshooting the hardware of DFP.

The ICT.TST file on the Z1800-series computer initiates dialog with DFP and steps through the worksheets; the worksheets send arguments to tell ptprog which test to run. On the DFP computer, the ptprog file contains the diagnostic commands and executes them.

No self-test fixture is needed for DFP self-test.

DR2p boards execute all ordinary Z1800-series diagnostics. Teradyne provides an additional DFP diagnostic that tests the CCCs and the DFP portion of the DR2p board.

To run the DFP diagnostic program:

1. **Open the 18XX "File" menu, then choose the \TPD directory and locate the "PTDIAGS" program.**
2. **Click on the "PTDIAGS" program.**

## Custom Application Developers

DFP applications are usually fairly simple and involve only small amounts (less than 100 lines) of code modification. DFP provides a number of working examples, fully explained, documented and instrumented with messages to the auxiliary CRT. Each of the examples can be modified to suit custom applications.

The usual DFP application consists of the following steps:

- recognize the command when it is issued by the main tester PC
- take control of the DR2p boards
- connect to the DUT
- verify silicon signature(s) of flash parts
- verify erasure; if not erased, erase
- write information, verifying each byte using an algorithm chosen according to part type
- disconnect from the DUT
- report result to main tester PC via serial port
- return to the waiting state

## Using DFP with Z1800-Series Testers

**Note: This section describes the use of Revision B.1 of the DFP software with Revision F.2a of the 18XX software.**

The 18XX interface provides setup and access to the DFP software. Through the Header variable, DFP, you can enable DFP, specify the source directory path, and specify communications channels AUX 1, 2, or 3. The DigFuncProc Device Type available in the Component Properties block of all sections except Interconnect enables you to generate a DFP worksheet.

**Note: You can have vector guards only in a DFP test that is in a digital section.**

### Setting Up DFP

#### COM Port Setup

Before you can use the DFP hardware, software, or run the PTDIAG program, the COM port must be set up within the 18XX environment.

1. **At the DOS prompt on the Z18XX keyboard, type <u>18XX</u> and press <Enter>**

#### Set Up The AUX Ports

1. **Select SETUP from the Main menu and press <Enter>.**
2. **Select DEVICE and CHANNEL DATA from the Setup menus and press <Enter>.**

   If AUX1 is not dedicated to any other functions, assign COM1 or COM2, as selected during the software installation, to AUX1.
3. **Select the AUX1 field and type COM1 or COM2 as appropriate.**

**Note: There must be <u>NO</u> spaces between COM and 1 or 2; for example, "COM  1" will not work.**

If AUX1 is in use, then you must assign the COM port to AUX2 (which will necessitate a number of changes to the PTDIAGS worksheets) or reassign the AUX1 functions to AUX2.

4. **Select the AUX2 field and type COM1 or COM2 as appropriate.**

**Note: To modify the PTDIAGS program to use AUX2 or AUX3, please contact Z-Series Product Support Group.**

5. **Select the Program Execute Channel block and set it to ON.**

Press the KEYPAD PLUS (+) key to display the Program Execute Channel pop-up menu.

Ensure that AUX1 (or AUX2 if used for Digital Function Processor) is checked (enabled).

6. **Select the Section Execute Channel block and set it to ON.**

   Press the KEYPAD PLUS (+) key to display the Program Execute Channel pop-up menu.

   Ensure that AUXI (or AUX2 if used for Digital Function Processor) is checked (enabled).

7. **Select the Step Execute Channel block and set it to ON.**

   Press the KEYPAD PLUS (+) key to display the Program Execute Channel pop-up menu.

   Ensure that AUX1 (or AUX2 if used for Digital Function Processor) is checked (enabled).

8. **Verity that the selected \TPD directory is C:\TPD**
9. **Select OK at the bottom of the screen.**
10. **Select SAVE to save the changes you have made to the DATA fields.**

### Set DFP Global Reboot Timeout

Select the Setup menu, then Environment Variables, then DFP Reboot Delay. Enter 30 seconds. Then, from the main menu, select the setup menu and save.

On 18XX OS F.2, set the DFP Boot Envirorunent Variable to 30 (seconds) or more to allow time for the DFP to re-boot and enter Slave mode.

**Note: If DFP is run under Win95, you will need to time a reboot and use THAT time plus half as a safety margin.**

If the DFP Boot Environment Variable is set, the GFI entry for "PTBOOT" is unnecessary, or should be DISABLED or deleted.

## Set Up The Serial Port On The 18XX System

The most important aspect of serial setup is that both the 18XX system and the DFP system must operate using the **same** serial baud rate, parity, handshake, and stop bits.

If **either** the 18XX system **or** the DFP system will be running in a DOS window under Win95, you will need to set up to run at 9600 baud.

**Note: The transfer program called DSZ.EXE is *not* compatible with Win95. Therefore, file transfers are not enabled if either system has Win95 active. (see below for settings)**

If **both** the 18XX system and the DFP system will be running under plain DOS (or DOS reboot from Win95), you will be able to run at 57600 baud.

The 18XX OS, Revision F.2a and above, can be setup using the Setup / Serial menu OR can be overridden using a DFP.CFG file in C:\PT2.

The 18XX OS Revision F. lb and older **must** be setup using the Setup / Serial menu DEFAULT VALUES and be overridden usinga DFP.CFG file in C:\PT2.

The 57600 setup / serial menu defaults are:

**Baud Rate: 57600**

**Parity: No parity**

**Word Length: 8**

**Stop Bits: 1**

**Handshake: None**

**Input Delay: 0 milliseconds**

**Output Delay: 0 milliseconds**

The older OS versions expect these exact serial settings in order to run DFP. For these systems you will need to copy the DFP.CFG file from the DFP system's C:\PT2 directory into the 18XX system's C:\PT2 directory to override the serial settings to whatever you need.

Under 18XX OS F. lb and earlier, verify that the RTS/CTS handshake box is **not** checked on the Serial Port Setup Screen.

The 9600 setup / serial menu defaults are:

**Baud Rate: 9600**

**Parity: No parity**

**Word Length: 8**

**Stop Bits: 2**

**Handshake: None**

**Input Delay: 109 milliseconds**

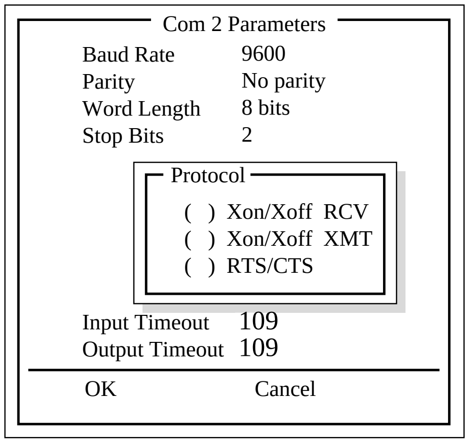
**Output Delay: 109 milliseconds**

Select **OK** at the bottom of the screen.

Select **SAVE** to save the changes you have made to the DATA fields.

Return to the 18XX Main Menu.

If either the Z1800 PC or the DFP PC is running Windows, the Com Parameters should match those shown below:



Com 2 Parameters

| | |
|---|---|
| Baud Rate | 9600 |
| Parity | No parity |
| Word Length | 8 bits |
| Stop Bits | 2 |

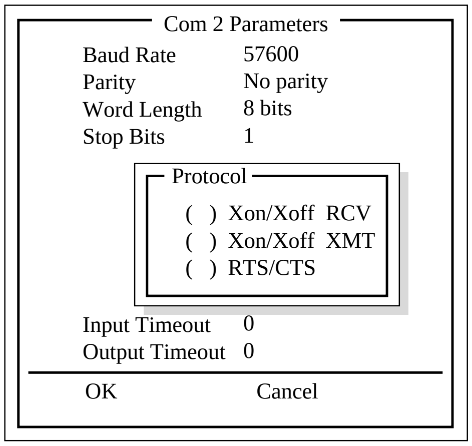
Protocol

( ) Xon/Xoff RCV
( ) Xon/Xoff XMT
( ) RTS/CTS

| | |
|---|---|
| Input Timeout | 109 |
| Output Timeout | 109 |

OK Cancel

If both the Z1800 PC and the DFP PC are running DOS, the Com Parameters should match those shown below:



Com 2 Parameters

| | |
|---|---|
| Baud Rate | 57600 |
| Parity | No parity |
| Word Length | 8 bits |
| Stop Bits | 1 |

Protocol

( ) Xon/Xoff RCV
( ) Xon/Xoff XMT
( ) RTS/CTS

| | |
|---|---|
| Input Timeout | 0 |
| Output Timeout | 0 |

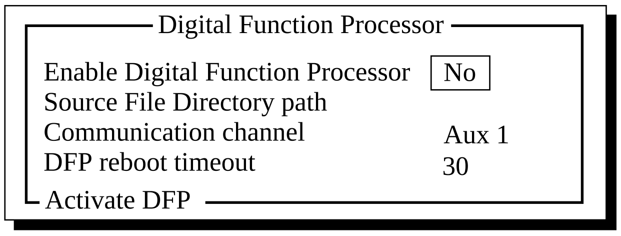
OK Cancel

## Setting up DFP in PRGMVARS

To set up DFP in the Header/PRGMVARS:

1. **Click on the DFP field in PRGMVARS/General Variables**

   The following window appears.



Digital Function Processor

| | |
|---|---|
| Enable Digital Function Processor | No |
| Source File Directory path | |
| Communication channel | Aux 1 |
| DFP reboot timeout | 30 |
| Activate DFP | |

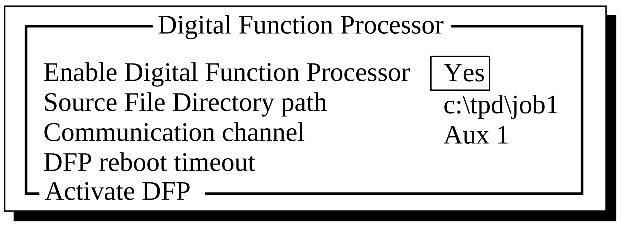
2. **Click on *No* in the Enable Digital Function Processor field to bring up the pop-up box, and select *Yes*.**
3. **In the Source File Directory path field, enter the complete path to the source directory path.**

   The path name is almost always the same as the board directory.
4. **In the Communication channel field, select the Auxiliary channel which contains the communications port for the DFP channel.**

   This should have been specified in the Setup/Data menu.

   The DFP reboot timeout should be set by manually rebooting, noting the time required for rebooting, then adding 50% to that time.

When you have finished setting up DFP, the DFP PRGMVARS window should look similar to the following.



Digital Function Processor

| | |
|---|---|
| Enable Digital Function Processor | Yes |
| Source File Directory path | c:\tpd\job1 |
| Communication channel | Aux 1 |
| DFP reboot timeout | |
| Activate DFP | |

## Generating the DFP Worksheet

To generate the DFP worksheet:

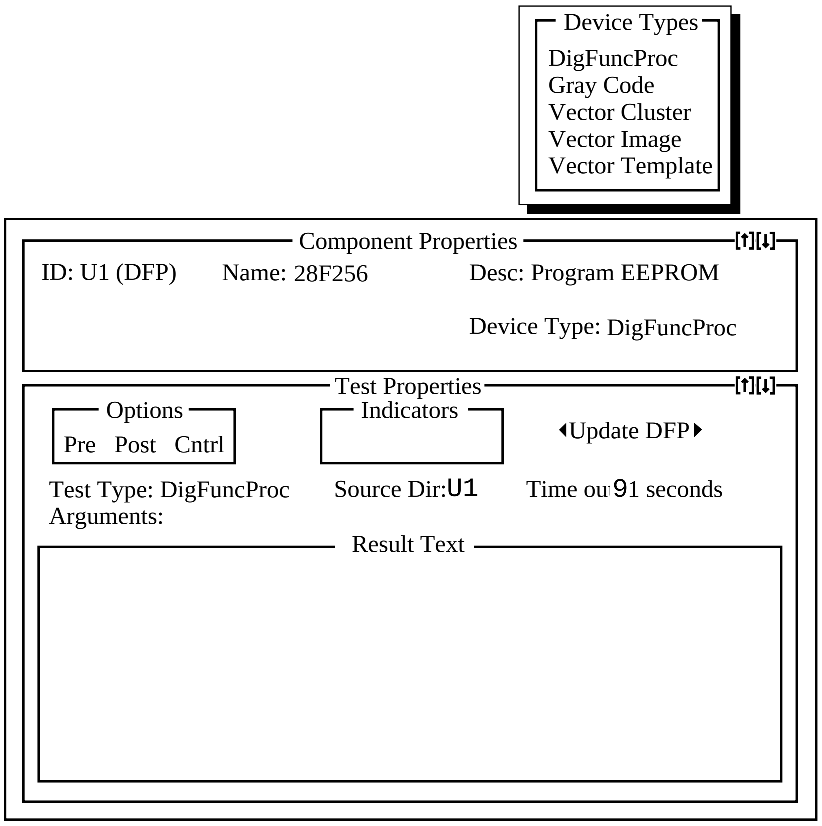
1. **In the test page Component Properties block, click on the Device Type field.**

   A pop-up window appears listing the available device types.

2. **Select *DigFuncProc*.**

3. **Select *Generate Test* from the Tools menu.**

   The Digital Function Processor worksheet appears. The following example shows a DFP test.



Device Types
DigFuncProc
Gray Code
Vector Cluster
Vector Image
Vector Template

Component Properties
ID: U1 (DFP)  Name: 28F256  Desc: Program EEPROM
Device Type: DigFuncProc

Test Properties
Options: Pre Post Cntrl
Indicators
Update DFP
Test Type: DigFuncProc  Source Dir: U1  Time ou 91 seconds
Arguments:
Result Text

## Worksheet Fields

The DFP worksheet fields enable you to manage DFP operation. The Result Text area displays messages sent back to the 18XX by PTPROG.EXE.

When you click on Update DFP, the most recent version of any file in the directory specified by the Source Dir field will overwrite identically named files on both computers. Update DFP will call DFPVER.EXE.

With DigFuncProc as the Device Type, the only Test Type available is DigFuncProc.

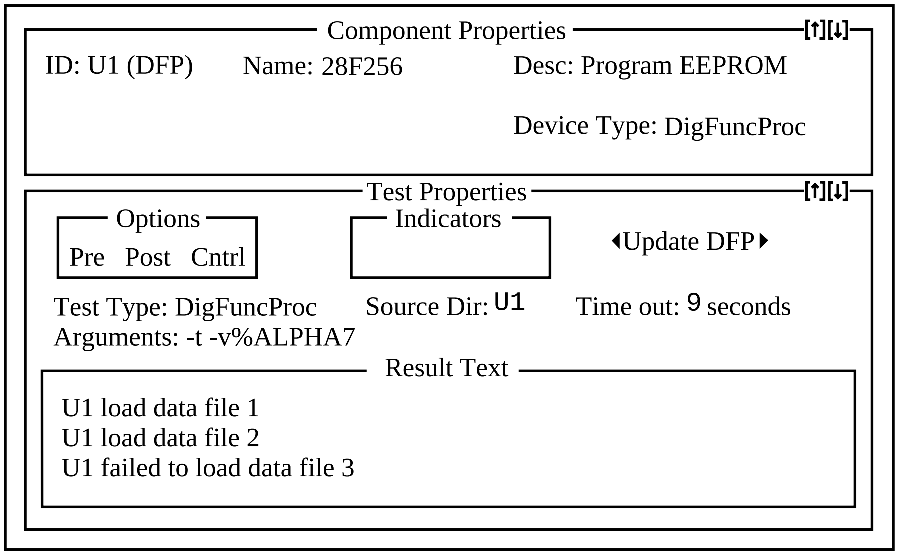
*Source Dir* refers to the subdirectory of the directory specified in the Header/PRGMVARS. The field takes a DOS directory name of eight characters. This directory contains the source files for the DFP test to be run from this worksheet. These files include PTPROG.EXE, PT2.INI, and any other files associated with this DFP test.

*Time out* enables you to specify the Com port timeout. If there is no activity on the Com port for the specified period of time, an error message is generated, and you will get a test failure.

The *Arguments* field enables you to send arguments to PTPROG.EXE.

After you have executed the DFP test, resulting messages from PTPROG.EXE appear in the Result Text box.

After your DFP test runs, the DFP worksheet will look similar to the following example.



ID: U1 (DFP)    Name: 28F256    Desc: Program EEPROM

Device Type: DigFuncProc

Test Type: DigFuncProc    Source Dir: U1    Time out: 9 seconds
Arguments: -t -v%ALPHA7

```
U1 load data file 1
U1 load data file 2
U1 failed to load data file 3
```

## Software Modules

Directory summaries for 18XX system software and DFP software are illustrated below.

Figure 2.2 DFP Directory Summary for Z1800-Series System Software

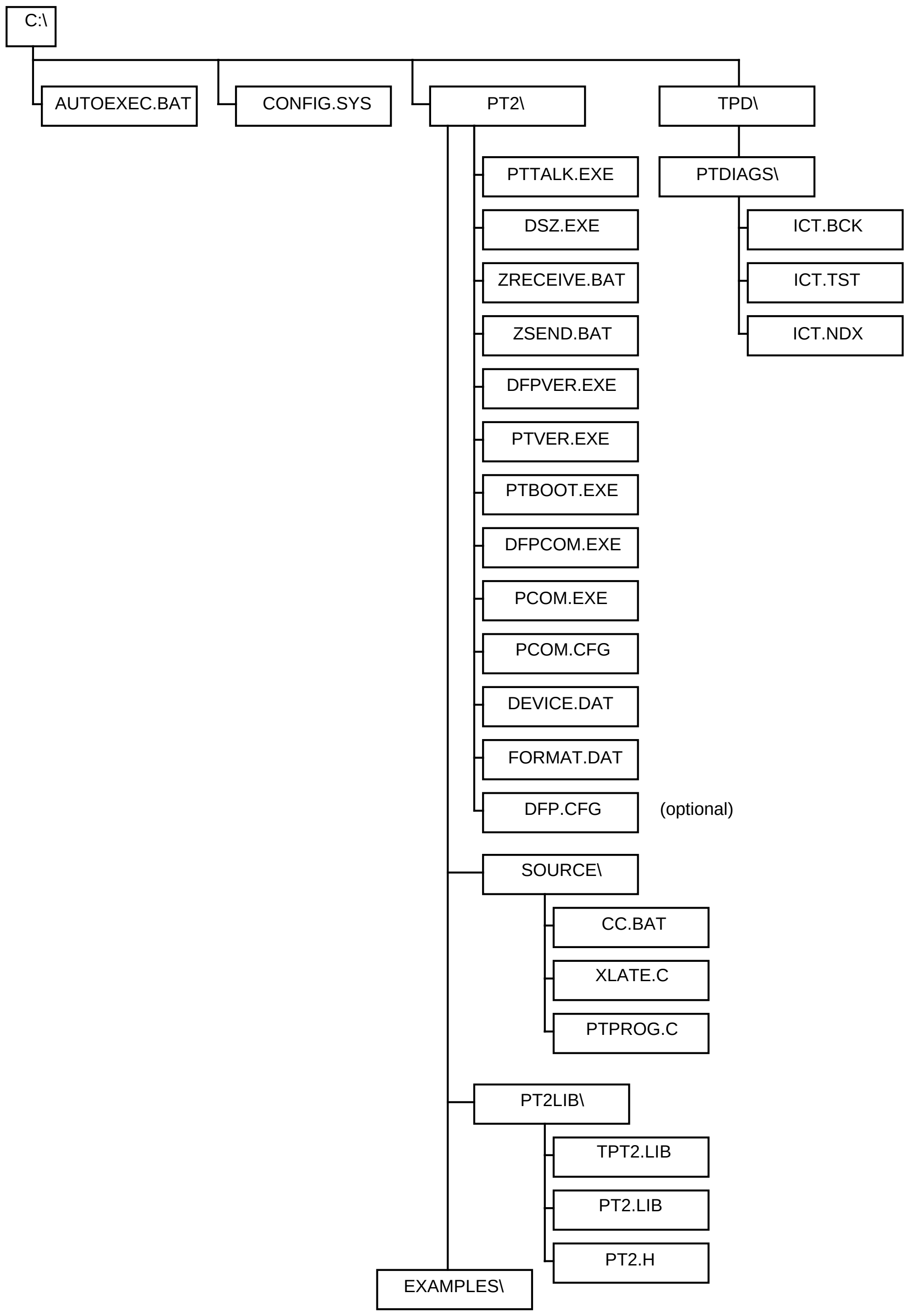
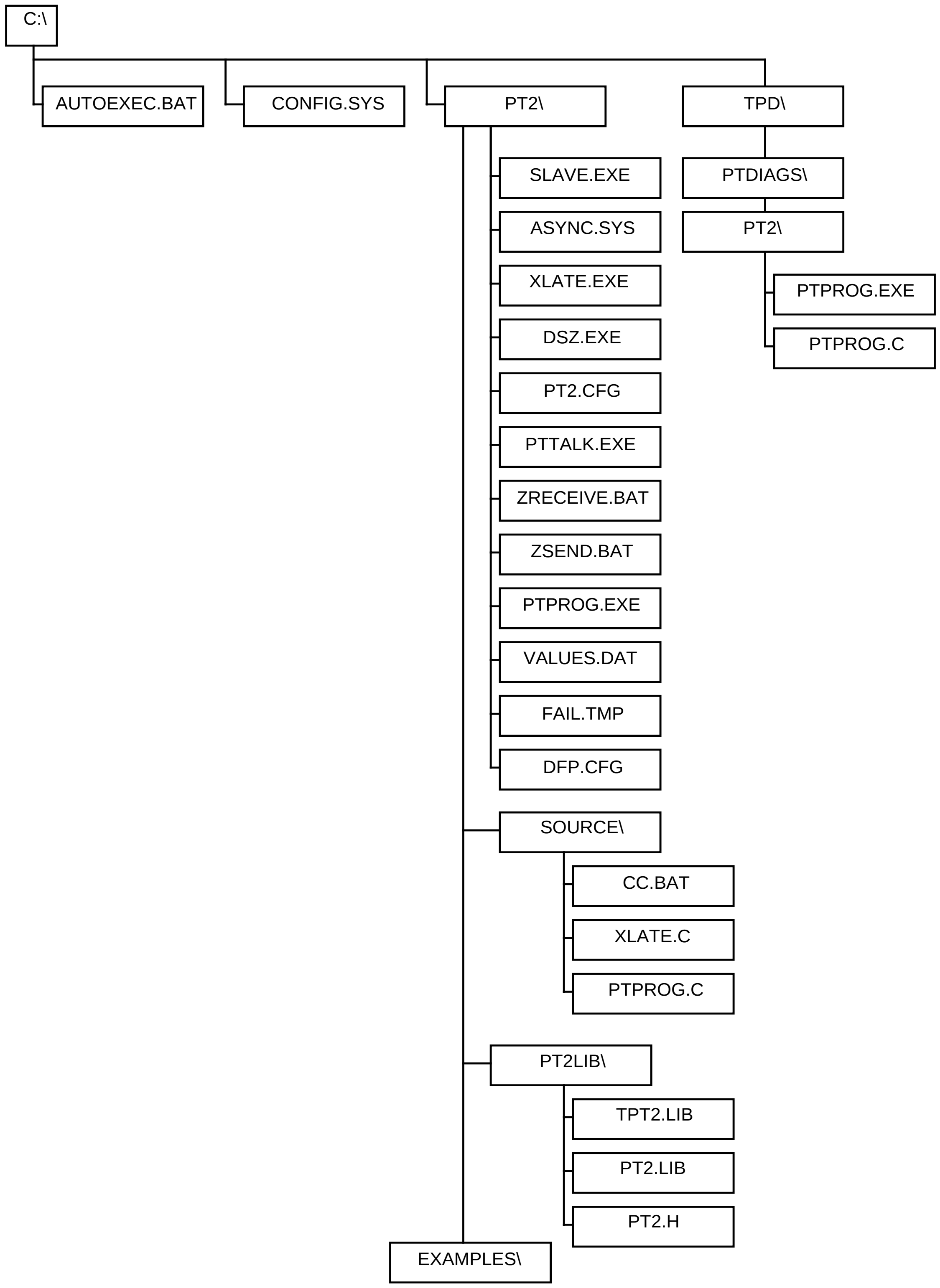


Figure 2.3 DFP Directory Summary for DFP System



## Standard Tools and Files

### SLAVE.EXE

(C:\PT2\SLAVE.EXE on the DFP system)

AUTOEXEC.BAT starts SLAVE.EXE on the DFP computer. You can also start SLAVE.EXE via the DFP keyboard from the command line. SLAVE.EXE's purpose is to interpret commands from the 18XX computer and perform functions which include job identification, DOS commands, file transfer and updates, file translations via XLATE.EXE, and algorithm execution.

SLAVE.EXE has no required arguments. After it is running, it accepts commands from DFP's serial port through a null modem cable.

To see SLAVE.EXE's optional arguments, type "SLAVE" or "slave" followed by a blank-space character, then press the Return or Enter key.

A command to SLAVE.EXE is an ASCII string consisting of an identifying character (uppercase only) followed by command arguments and terminated by a carriage return (0xd). Arguments are separated by spaces, and there is no space between the identifying character and the first argument.

Pressing Esc on the DFP computer's keyboard ends the execution of SLAVE.EXE. Restart SLAVE.EXE after you perform any DOS commands on the DFP computer.

The commands SLAVE.EXE will accept are listed below, then described in detail in the next section.

- **V (Version)**
- **D (Date and Time)**
- **J (Job or board-level directory)**
- **S (Send module directories)**
- **R (Receive module directories)**
- **P (Program - start PTPROG)**
- **F (Failure flags)**
- **L (Location of other data)**
- **C (Configuration)**
- **X (Execute DOS command)**
- **Q (Quit)**
- **I (Initialize DR2Ps)**
- **T (Transfer files check)**
- **M (Master OK)**

### Command Descriptions for SLAVE.EXE

**V command:** Ensures the software on the 18XX computer is the same revision as that on the DFP. DFPVER.EXE issues the command from an 18XX in-circuit PRGMVARS header test step. This command has no arguments. SLAVE.EXE returns the ASCII string defined as VERSION in PT2.H.

**D command:** Sets the date and time on the DFP computer. DFPVER.EXE issues the command and sets the date and time to match the 18XX computer. This command has six arguments and all must be present with one space between them:

```
D<year> <month> <day> <hour> <min> <sec>
```

Example:

```
 D1994 04 15 13 30 12
```

SLAVE.EXE returns the following values defined in PT2.H:

| | |
|---|---|
| DATE_ERROR | if date could not be set. |
| TIME_ERROR | if time could not be set. |
| DATE_SET | if command was successful. |

**J command:** Tells SLAVE.EXE where the module directories reside. Multiple modules can be associated with an in-circuit test program. This path is duplicated on the DFP computer. The full path name is needed, including the drive. The path is not required to be the same as the path for the in-circuit test. Drive names are usually set up during installation of the DFP software; if none are set up during installation, only drive C: is available. Currently DFP supports drives C through F. These are directories substituted on the C drive. More can be added as needed.

Example:

```
Jc:\tpd\ptdiags
```

Slave returns the following values defined in PT2.H:

| | |
|---|---|
| SLAVE_OK | if SLAVE.EXE successfully created the directory. |
| INVALID_DRIVE | if the drive does not exist. |
| INVALID_JOBPATH | if SLAVE.EXE was unable to change to the job path directory. |

**S command:** Starts the process for SLAVE.EXE to send module directory contents to the 18XX computer. You can add an optional V (verbose) to send ZMODEM standard error messages to the DFP monitor. DFPVER.EXE issues the S command from a Header test step in the in-circuit test program. SLAVE.EXE returns the following value defined in PT2.H:

| | |
|---|---|
| SLAVE_OK | SLAVE.EXE received command. |

SLAVE.EXE now searches the job directory for all module subdirectories. It is assumed that only DFP-related information resides on the DFP computer. Upon finding a directory, slave sends the 18XX computer an “S” followed by the directory name. SLAVE.EXE then changes to the module directory and starts zsend.bat. After zsend.bat is complete, a sync is necessary to ensure the buffers are flushed in case ZMODEM did not finish cleanly. This involves both sides trading a single character “H” until a predetermined number is read. Both computers then send a terminating character “I”. The port is then read until the terminating character is found. SLAVE.EXE now changes back to the job directory and continues the search. When all directories have been transferred, a single S is sent to inform DFPVER.EXE that SLAVE.EXE is finished. SLAVE.EXE returns the following values defined in PT2.H:

| | |
|---|---|
| SLAVE_OK | all is well. |
| DSZ_ERROR | unable to complete ZMODEM transfer. |

**R command:** Starts the process for SLAVE.EXE to receive module directory contents from the 18XX computer; it is almost the same as the S command described previously. DFPVER.EXE sends SLAVE.EXE an R command followed by a module directory name. A V can be appended for ZMODEM verbose mode. SLAVE.EXE will create this directory, change to it, and start ZRECEIVE.BAT. A sync is performed and XLATE.EXE is started in this directory. Any needed translation is done at this time. SLAVE.EXE returns the following values defined in PT2.H:

| | |
|---|---|
| SLAVE_OK | all is well. |
| INVALID_VERSION | unable to change to this directory. |
| INVALID_JOBPATH | unable to change to this directory. |
| XLATE_DONE | translate complete. |
| XLATE_ERROR | error translating files. |
| DSZ_ERROR | unable to complete ZMODEM transfer. |

**P command:** Sent to SLAVE.EXE from an in-circuit test step of type DigFuncProc. The P is followed by a module directory and any arguments to pass to PTPROG.EXE. SLAVE.EXE then changes to the module directory and starts PTPROG.EXE. PTPROG.EXE is started with “-s” as the first argument, telling ptprog that it was started by slave and not from the command line. After ptprog finishes, slave opens the file c:\pt2\fail.tmp and reads an integer from this file. This integer is the failure flag that the F command (see below) returns. SLAVE.EXE immediately removes the fail.tmp file. SLAVE.EXE returns the following values defined in PT2.H:

| | |
|---|---|
| P_DONE | ptprog exited error-free. |
| PTPROG_ERROR | slave was unable to start ptprog. |

A PTPROG.EXE exit status of zero is considered error-free. If PTPROG.EXE exits with a nonzero, SLAVE.EXE returns that value. There are several defined error codes associated with PTPROG.EXE; these are included in the listing of PT2.H in a later chapter.

**F command:** Sent by PCOM.EXE (the predecessor to DFPCOM.EXE) to request the failure flags set by the last PTPROG.EXE to run. The flags are an integer that was previously read from the file c:\pt2\fail.tmp.

**Note: The F command is not used with the 18XX DigFuncProc worksheet.**

**L command:** Provides a means to send SLAVE.EXE data that may not be available when the P command is sent. %MEAS data from a capacitor test is one example. The sixteen locations (1-16) available to store this data are identified by a number following the L and then the data. The complete string (including the L) is saved to a file called c:\pt2\values.dat . Each string can be up to 600 characters in length. The first location is stored at the beginning of this file and the rest are stored at multiple offsets of 600.

"L[Name] <arg> <arg>" where [Name] is an integer from 1 through 16.

Example:

```
L1 %MEAS
```

**Note: The L command cannot be sent by the 18XX DigFuncProc worksheet; it is generally sent via an I/O string in the pre/post options of a regular 18XX worksheet.**

**C command:** Sent by DFPVER.EXE. Upon receiving this command, SLAVE.EXE opens the configuration file C:\PT2\PT2.CFG. SLAVE.EXE reads the configuration string and returns it.

**X command:** Executes any DOS command following. This command can be sent to SLAVE.EXE from PTTALK.EXE. SLAVE.EXE always returns the value INVALID_COMMAND, defined in PT2.H. Use caution with this command.

**Q command:** Terminates SLAVE.EXE. Before terminating, SLAVE.EXE will return the value TERMINATE defined in PT2.H.

Invalid commands sent to SLAVE.EXE cause it to return the value INVALID_COMMAND defined in PT2.H.

**I command:** Initializes the DFP Channel Control and DR2p cards.

**T command:** Get file transfer options. This command causes the DFP to send the file transfer options, stored in the DFP.CFGfile, to the 18XX system.

**M command:** Displays “Status = Master OK” on the DFP Terminal.

## XLATE.EXE

(C:\PT2\XLATE.EXE on the DFP system)

XLATE.EXE is started from SLAVE.EXE. on the DFP computer. You can also start it from the DFP command line using the keyboard.

XLATE.EXE translates your data files into a form directly usable by DFP. XLATE.EXE accepts Motorola S-records or Intel hex records, the two most common PROM programmer formats, and other translation types (detailed in the “format.dat” section later in this chapter). “No-translation” is also acceptable.

XLATE.EXE checks the date of data and image files to determine if updating the translated files is necessary. XLATE.EXE will translate the data record(s) listed in PT2.INI file. It gives files it creates the same name as the data record file, with the extension “.img” replacing the data record extension. The data record is checksummed, sorted, and translated into a binary image file with the address holes filled appropriately. XLATE.EXE updates the image file date and time to match the data record file.

**Note: The translation types are detailed in the “FORMAT.DAT” section on page 2-32; board addressing is discussed in the “PT2.INI” section on page 2-31.**

## DSZ.EXE

(C:\PT2\DSZ.EXE on the DFP system)

DSZ.EXE is a commercial ZMODEM program from Omen Technology. It implements the ZMODEM file transfer protocol. DSZ features the ZMODEM-90™ extensions including ZMODEM compression and MobyTurbo™ accelerator.

**Warning: DSZ.EXE is not compatible with Windows95, so file transfers are not available under Windows95’s DOS shell.**

See page 2-23 for information about DFP.CFG's TRNSFER_FILES and TRNSFER_WIN_FILES values.

The ZMODEM file transfer protocol provides reliable file and command transfers with complete END-TO-END data integrity between application programs. DSZ's 32-bit CRC protects against errors that are not detected by both "error free" modems and the most advanced networks.

## PTTALK.EXE

(C:\PT2\PTTALK.EXE on both the DFP and 18XX computers)

PTTALK.EXE is started with a command from the Z1800-series computer keyboard.  It enables a communications link to the SLAVE.EXE via the Z1800-series keyboard; thus it accepts such commands as V (ver), D (date), and J (job path). It is intended as a debug tool only. Pressing Esc or Ctrl-C ends a PTTALK.EXE session.

PTTALK.EXE can be started on the 18XX computer to verify communication; typing "PTTALK.EXE ?" will display the command usage.

To verify communication between the machines using PTTALK.EXE:

1. **Make sure that SLAVE.EXE is running on the DFP computer.**
2. **At the 18XX system's DOS prompt, enter *pttalk comX*, where "X" is the com port number (such as 1 or 2), then press Enter.**
3. **At the 18XX computer, type V , then press Enter.**

   The current DFP software version will be returned and displayed on the 18XX screen (see the section covering the SLAVE.EXE V command earlier in this chapter).
4. **Press Esc to exit.**

## PTPROG.EXE

(C:\PT2\PTPROG.EXE and C:\TPD\PTDIAGS\PT2\PTPROG.EXE on the DFP computer)

The DigFuncProc worksheet on the 18XX sends the P command (described earlier in this chapter) to SLAVE.EXE. SLAVE.EXE then starts PTPROG.EXE on the DFP computer. Multiple ptprogs can exist; the one that runs depends on file hierarchy - directory, path, and so on.

PTPROG.EXE is responsible for a number of activities, including loading nonvolatile memory with variable data or with data stored in an image file. It also passes error messages and the pass/fail information to the DigFuncProc worksheet; see also the section covering DFPCOM.EXE later in this chapter.

As a program written by the user, PTPROG.EXE can easily be customized for a particular application. Tasks that can be specified in PTPROG.EXE include:

- Opening com ports
- Getting and checking program arguments
- Opening and reading PT2.INI files
- Opening and reading data files
- Setting up CCC/DR2P cards
- Performing device operations appropriate to the application (erasing, programing, verifying, and so on)
- Writing pass/fail data to 18XX
- Closing all files and ports
- Releasing DRs
- Exiting the test

A PTPROG.EXE exit status of zero is considered error-free. If PTPROG.EXE exits with a nonzero status, SLAVE.EXE returns that value. There are are several defined error codes associated with PTPROG.EXE; these are included in the listing of PT2.H in a later chapter.

**Note: See the examples for custom applications later in this manual; others may be available from Teradyne.**

### ZRECEIVE.BAT

(C:\PT2\ZRECEIVE.BAT on both the DFP and 18XX computers)

Used only under MS/DOS, not under Windows operating systems.

This DOS batch file contains the necessary commands and arguments to receive files using ZMODEM file transfer protocol. This file is used on both the 18XX computer and the DFP computer.

### ZSEND.BAT

(C:\PT2\ZSEND.BAT on both the DFP and 18XX computers)

Used only under MS/DOS, not under Windows operating systems.

This DOS batch file contains the necessary commands and arguments to send files using ZMODEM file transfer protocol. This file is used on both the 18XX computer and the DFP computer.

## ASYNC.SYS

(C:\PT2\ASYNC.SYS on the DFP computer)

Async.sys is a device driver that manages the serial ports on the PC and enables interrupt driven and buffered communication. Simple four or eight port extension boards (e.g., digiboard) can be used in conjunction with the standard com1 and com2 hardware usually found in PCs. Interrupts are shared among the ports on one board; i.e., it is not possible to have the standard com2 and a digiboard share interrupt 3. The driver supports up to eight devices.

The driver installs through the CONFIG.SYS interface on both the 18XX computer and the DFP computer and has the following command line syntax:

```
device=async.sys COM:n,i,p,[ibuf],[obuf]; COM:n,I,P,[ibuf],[obuf]; ...etc
```

where

- •n is the port number in the range of 1 to 8
- •i is the IRQ number in the range of 2 to 7
- •p is the base port address in hex
- •ibuf is the input buffer
- •obuf is the output buffer
- •spaces are optional to improve readability

**Note: The buffer sizes are not optional for DFP. The values of 1024 for the input buffer and 1024 for the output buffer must be used with DFP.**

## PT2.CFG

(C:\PT2\PT2.CFG on the DFP system)

PT2.CFG is the predecessor to DFP.CFG.

## DFP.CFG

(C:\PT2\DFP.CFG on the DFP system)

DFP.CFG contains system/DFP configuration and serial port information.

DFP.CFG is created on the DFP computer during installation of the DFP system; it can optionally be created on the Z18XX system. DFP.CFG should include which CCC boards are installed in the DFP computer and which node numbers (DR2P cards) are associated with each CCC. The entry in the file is one line in a group labelled “[General]”.

Example:

```
CCC=0,192,1,224
```

The connection from the CCC to the DR2p board is defined by the first node number on the DR2p. CCC 0 is connected to the DR2p at node 192. CCC 1 is connected to the DR2p at node 224.

If you add DR2p’s after installation, or if you move any DR2p’s to new locations in the test head cage, you should edit PT2.CFG to match the new current status of the DFP system.

Example DFP.CFG file:

```
[Serial]
   SerVerbose=YES   # else NO - output DFP.CFG serial settings?
                    # (Default = Yes)
   SerBaud=9600     # else 110,...,9600,19200,38400,57600
                    # (DOS Default = 57600, WINDOWS Default = 9600)
   SerParity=None   # else Odd, or Even (Default = None)
   SerDataBits=8    # else 5,6, or 7 (Default = 8)
   SerStopBits=2    # (DOS Default = 1, WINDOWS Default = 2)
   SerProtocol=NONE # else XRCV,XXMT,XALL,HNONE,HXRCV,HXXMT,HXALL
                    # (Default = NONE (ie uses RTS/CTS)
   SerIwait=100ms   # else 0-10000ms (resolution is 50ms)
                    # (Default = l00ms)
   Ser0wait=100ms   # else 0-10000ms (resolution is 50ms)
                    # (Default = l00ms)
[General]
   GenVerbose=Yes   #else NO-output dfp,cfg general ettings?
   CCC=0,192,1,224              # else any combination of
                                # <ccc number>,<start node>.
                                # Note: ccc #'s should start
                                # from 0 and be consecutive.
   TRNSFR_FILES=Yes             # Else No.
                                # If No - Don't transfer files
                                # during updateDFP.
   TRNSFR_WIN_FILES=No          # If No - Don't transfer files
                                # during updateDFP if either system
                                # is running Windows(OS = WINDOWS)
```

## VALUES.DAT

(C:\PT2\VALUES.DAT on the DFP system)

VALUES.DAT is created by SLAVE.EXE. Values.dat resides in the pt2 directory and contains the arguments/data that are sent to SLAVE.EXE via the L command. This data is accessible to all programs that reside on the DFP computer.

## FAIL.TMP

(C:\PT2\FAIL.TMP on the DFP system)

If PTVER.EXE and an old-style mixed-mode worksheet are in use, Fail.tmp is created by PTPROG.EXE. It contains a failure flag integer and is used to return failure information to DFPCOM.EXE. SLAVE.EXE reads and deletes fail.tmp after PTPROG.EXE returns control to SLAVE.EXE. The failure flag integer is the value returned to DFPCOM.EXE when the F command is sent to slave. Each binary bit in this integer represents a flag. A failure is bit = 1.

## CC.BAT

(C:\PT2\SOURCE\CC.BAT on both the DFP system and the 18XX system)

Included in the DFP documentation is a sample batch file for command line compiling for Borland Turbo C, Microsoft Quick C, and Microsoft Visual C/C++.

It can be found on the DFP computer under pt2\source. Copy the file to whatever directory you need for compiling your ptprog.c.

To use the batch file, simply “rem” the libraries you DO NOT wish to use, then delete the “rem” for the library you DO wish to use (However, leave the comment for identifying compiler (library) with a rem).

The example below is for use with Turbo C:

```
@rem Borland Turbo C
tcc -ms -a- -f -2 -DTURBO -Ic:\pt2\pt2lib -Lc:\pt2\pt2lib %1.c tpt2.lib

@rem Microsoft Quick C
@rem qcl /Zp /AL /FPi /W3 /G2 /c %1.c
@rem qlink /NOD /ST:4096 %1.obj,,NULL,pt2+llibce

@rem Microsoft Visual C/C++
@rem cl /Zp /AL /FPi /W3 /G2 /c %1.c
@rem link /NOD /ST:4096 %1.obj,,NULL,pt2+llibce+oldnames,,
```

**Using the Turbo C environment on the DFP computer:**
If you opt to use the Turbo C library and have loaded it on your DFP computer, you may wish to use the TC environment to debug your ptprog.c. To use the environment:

1. **Edit the AUTOEXEC.BAT, removing the smartdrv extensions.**
2. **Reboot the DFP computer.**

3. **Enter the TC environment.**
4. **Under "Options:Directories:Include" add *C:\PT2\PT2LIB***
5. **Under "Options:Directories:Libraries" add: *C:\PT2\PT2LIB***
6. **Under "Options:Compiler: Code Generation" add to Define: *TURBO***
7. **Under "Project" open a project.**
8. **Add to your project the PTPROG.C file and the TPT2.LIB file.**
9. **Under "Run: Arguments" add *-k*, *-p*, or any arguments needed to run your PTPROG.EXE from the DFP keyboard.**

   Now you are ready to open the PTPROG.C file and run and edit as needed.

## PT2.LIB and TPT2.LIB

(C:\PT2\PT2LIB\PT2.LIB and TPT2.LIB on both the DFP system and the 18XX system)

The library files are a collection of functions used for development of the DFP system. They can be used to create a custom DFP application and include functions for communication with the channel control card/driver receiver board, the async driver, and the DFP system. The functions are listed and described in the PT2.H listing included in a later chapter of this manual. The paths are pt2\pt2.lib and pt2\tpt2.lib on either the DFP or 18XX computer, depending on the options you chose at installation.

Pt2.lib is compiled for use with Microsoft C-language products, and tpt2.lib is compiled for use with Borland Turbo C-language products.

## PT2.H

(C:\PT2\PT2LIB\PT2.H on both the DFP system and the 18XX system)

Pt2.h is a header file that contains the prototype declarations for the DFP system. Include PT2.H in your custom program in order to use the DFP library.

**Note: The content of the PT2.H file appears in the "PT2.H Listing" chapter of this manual.**

## DFPVER.EXE

(C:\PT2\DFPVER.EXE on the 18XX system)

DFPVER.EXE is started from the 18XX program Header via pgmvars. It will run automatically the first time you enter the 18XX program in Run mode.

DFPVER.EXE creates the file C:\PT2\DFPCOM.CFG which dfpcom uses to identify the com port to use. It also sends several basic commands to SLAVE.EXE on the DFP computer. It opens communications to DFP via SLAVE.EXE, checks on the version of SLAVE.EXE, and allows job identification and file transfers. Once all the files on the 18XX are updated to the latest versions, DFPVER.EXE terminates, and the 18XX program continues.

If SLAVE.EXE does not respond or receives an incorrect response when communications are first opened, DFPVER.EXE will reboot the DFP computer and try one more time. DFPVER.EXE can also be run from the DigFuncProc worksheet via the UPDATE field when in 18XX debug mode; the worksheet's UPDATE field calls this function.

## PTVER.EXE

(C:\PT2\PTVER.EXE on the 18XX system)

PTVER.EXE is the predecessor to DFPVER.EXE.

## PTBOOT.EXE

(C:\PT2\PBOOT.EXE on the 18XX system)

PTBOOT.EXE is started from a cold boot via the Z1800-series system software's AUTOEXEC.BAT or from the Z1800-series system's keyboard. After PC I/O has been loaded, it can also be called via dfpver if the DFP computer does not respond to DFPVER.EXE.

PTBOOT.EXE toggles a built-in system relay which causes a hardware reset of the DFP computer. On some DFP systems, one of the RAB option relays is used. On other systems, the power supply controller provides the DFP reset relay.

If you start PTBOOT.EXE with no arguments, PTBOOT.EXE announces itself and there is a 30-second delay before PTBOOT.EXE exits. Any arguments given to PTBOOT.EXE will cause silent operation and no delay.

Occasional crashes can be expected in an embedded system running customer-developed code, especially during program development.  DFP's cold boot facility provides a defense against this inconvenience.

If, on the initial run of the in-circuit board test program, dfpver in the Z1800-series computer cannot establish communication with slave in the DFP computer, it presumes a crash or other software hangup. Dfpver then executes ptboot.

The custom programmer should identify and correct the causes of such crashes prior to putting the custom software in service. DFP's CRT facility allows you to connect a display device, which the programmer can use to identify the custom software's progress prior to the crash.

## DFPCOM.EXE

(C:\PT2\DFPCOM.EXE on the 18XX system)

DFPCOM.EXE is automatically initialized via the DigProcFunc 18XX worksheet; it is responsible for passing and receiving information between the 18XX DigProcFunc worksheet and the DFP computer. DFPCOM.EXE reads the file C:\PT2\PCOM.CFG for the com port to use, then initiates communications with SLAVE.EXE running on the DFP.

DFPCOM.EXE passes to SLAVE.EXE the source directory (board subdirectory where the PTPROG.EXE actually resides) found in the source field of the DigProcFunc worksheet. Any arguments found in the argument field of the DigFuncProc worksheet are also passed to SLAVE.EXE at this time.

DFPCOM.EXE listens for and processes several types of messages.

### User-Programmable Messages

There are three types of messages/define statements the user can program into the ptprog.c program:

**KEEP_ALIVE:** The DFPCOM.EXE can timeout. The timeout (in seconds) is set by the user in the Timeout field of the DigFuncProc worksheet. The default timeout is 9 seconds. PTPROG.EXE running on the DFP computer may send a KEEP_ALIVE message to DFPCOM.EXE, resetting the timer. Should dfpcom timeout, the test step will issue an 18xx Test Step Error instead of an 18xx Failure. This message is sent to DFPCOM.EXE from the PTPROG.EXE test via the pt2lib function keep_alive():

```
keep_alive(KEEP_ALIVE);
```

**PASS/FAIL:** The PASS or FAIL message is passed to DFPCOM.EXE from the PTPROG.EXE running on the DFP computer. This will be relayed to the 18xx program once P_DONE is received from SLAVE.EXE (see P_DONE below). This message is sent to DFPCOM.EXE from the PTPROG.EXE test via the pt2lib function keep_alive():

```
keep_alive(handle,PASS);        or

keep_alive(handle,FAIL);
```

**DFP_18XX_MSG:** The DFPCOM.EXE can receive message strings from the PTPROG.EXE running on the DFP. These messages can be used for a variety of purposes, including debugging ptprog.c programs or indicating which device failed if more than one device is being tested. The message strings are displayed in the Result Text field of the DigFuncProc worksheet in 18xx debug mode. The messages can also be printed to the CRT and printer during 18xx run mode if the Test Page All print option is enabled under the DigFuncProc worksheet Cntl option. This message is sent to DFPOM.EXE from the PTPROG.EXE test via the pt2lib function send18xxMsg():

```
send18xxMsg(handle,DFP_18XX_MSG,“U1 FAILED”);
```

**Note: Do NOT CONFUSE this message string with the PASS/FAIL message above. This is a message string handler only, not the method to indicate pass/fail to the 18xx program.**

**DFP_18XX_ERR_MSG:** This message string is a variant of the “DFP_18XX_MSG” message described above; however, the string is displayed briefly in a little red box overlaid on the DigFuncProc page, instead of in the Result Text field. Also these messages CAN NOT be printed out or saved, and therefore they are ONLY meant for debugging the user PTPROG.EXE program, and are NOT intended for passing errors from the final PTPROG.EXE program that the user wants printed out if the teststep fails.

```
send18xxMsg(handle,DFP_18XX_ERR_MSG,“Can’t open data file”);
```

**Note: This error message will NOT end the DigFuncProc test. This is a message string handler only. An error indication to end the DigFuncProc test must come via the SLAVE.EXE to ensure SLAVE.EXE becomes active again and the DFP itself hasn’t hung.**

### Automatic messages

Two other types of messages are taken care of by the SLAVE.EXE program when PTPROG.EXE terminates. These messages are NOT meant for the user to program:

**P_DONE:** When PTPROG.EXE terminates, SLAVE.EXE will pass the P_DONE message to DFPCOM.EXE indicating that the PTPROG.EXE successfully returned control to SLAVE.EXE and consequently the DFPCOM.EXE will now pass the PASS/FAIL information to the 18xx program and the 18xx program will continue.

**PTPROG_ERROR:** If the PTPROG.EXE program terminated early with an error message, SLAVE.EXE will pass the error message to DFPCOM.EXE, indicating PTPROG.EXE successfully returned control to SLAVE.EXE. DFPCOM.EXE will briefly display an error box on the 18xx screen with either a defined message or a message number, then terminate the 18xx test step.

**Note: If the user wants to print the error, the user needs to send the DFP_18XX_MSG (NOT the DFP_18XX_ERR_MSG - see both above) just before the PTPROG.EXE calls either exit() or uses a return (to SLAVE.EXE) with the error.**

## PCOM.EXE

(C:\PT2\PCOM.EXE on the 18XX system)

PCOM.EXE is the predecessor to DFPCOM.EXE.

## DFPCOM.CFG

(C:\PT2\DFPCOM.CFG on the 18XX system)

The file DFPCOM.CFG is created by DFPVER.EXE when the 18XX test program is run. It contains the com port argument given to DFPVER.EXE and is used by the DigFuncProc worksheet (via DFPCOM.EXE) to identify the com port to use when communicating with the DFP computer. It also contains error/no-error indications of successful execution of DFPVER.EXE.

Example: COM1 noerror

## PT2.INI

(C:\TPD\JOB\MOD\PT2.INI)

A PT2.INI file is used to identify a subdirectory as a DFP directory containing DFP test files. The PT2.INI file contains information used by XLATE.EXE and by the application’s PTPROG.EXE. The entries are optional depending, on the application’s need; however, the most commonly used entries are the L, M, and R entries: L = Local device, M = Manufacturer code & id, and R = Remarks (user remarks and information). Below is a PT2.INI file example, followed by explanations for each entry.

Example PT2.INI file:

```
L,U1,28F010,u1.dat,87,131072,0,1
M,89,B4
R, format 87 = Motorola S record type
R, u1.dat update 3-2-95
```

**LOCAL entry**: The L entry is for a specific device. One PT2.INI file can have several L entries, indicating the devices are to be tested together, as in the case of a common address bus. For custom work only the first eight fields need to be entered, with each field separated by a comma.  These first eight fields are:

1: L (tag for device entry)
2: device id name
3: actual device type
4: data record filename
5: data record format type
6: memory size
7: bus position
8: fill undefined locations with either 0's or 1's

Example: L,U1,28F010,u1.dat,87,131072,0,1

L = local tag
U1= customer chip ID
28F010 = device type is a 28F10
u1.dat = data record filename is 66e.ptp
87 = data record format type is Motorola S record
131072 = memory size
0 = bus position is 0
1 = fill undefined locations with 1

**MANUFACTURER entry:** The manufacturer code and manufacturer id are entered in sets of two. The entry covers all possible sets allowed  for the devices entered in the PT2.INI file. The minimum number of fields for a M-tag entry is three fields.

1: M (tag for manufacturer code & id entry)
2: manufacturer code
3: manufacturer id
4: manufacturer code
5: manufacturer id  , and so on...

Example: M,89,B4

M = manufacturer tag
89 = manufacturer code (hex) for intel 28F010
B4 = manufacturer id (hex) for intel 28F010

**REMARK/S entry:** Remarks are entered to allow user information to be recorded. There can be multiple R-tag entries in a PT2.INI file.

1: R (tag for remark/s)
2: remark string

Example:

R, format 87 = Motorola S record type R, u1.dat update 3-2-95

## FORMAT.DAT

(C:\PT2\FORMAT.DAT on the 18XX system)

FORMAT.DAT documents the allowable data record formats and contains data record format information used by XLATE.EXE.

FORMAT.DAT has two fields for each entry. The fields are separated by a comma, with no blank spaces.

Example:

```
Motorola_Exormax,87
```

In the example above, Motorola_Exormax is the data record format name and 87 is the data record translation format code.

XLATE.EXE currently allows the following formats:

```
No_translation,0
Motorola_Exorciser,82
Intel_Intellec_8/MDS,83
Motorola_Exormax,87
Intel_MCS-86_Hex_Object,88
Jedec_format(Full),91
Jedec_format(Kernel),92
Motorola_32bit(S3_record),93
```

## File Maintenance

A Z1800-series tester with DFP consists of two computers and two hard disks. To keep the information current in both hard disks, DFP automatically runs, as needed, a software system based on the commercial ZMODEM standard (as described in the section covering the DFPVER.EXE program, above).

**Note: This is not true when DFP or 18XX are operating under Windows.**

The ZMODEM software ensures that files updated in one computer are automatically updated in the other. Thus if a file relevant to DFP is updated in the main PC, DFP automatically receives the updated copy the next time it is needed. By the same token, files edited on the DFP computer during custom development will be automatically copied into the appropriate place on the main PC when commands involving them are executed.

Because of this feature, it is not necessary to make regular backups of the DFP computer's hard disk. Backing up the host Z1800-series computer also creates backups of the DFP user files.

## Diagnostic System

The diagnostic system consists of TPD\PTDIAGS\ICT.TST on the Z1800-series computer, and TPD\PTDIAGS\PT2\PTPROG.EXE on the DFP computer.

The ICT.TST file on the Z1800-series computer automatically initiates dialog with DFP and steps through the worksheets. The worksheets send arguments to tell PTPROG.EXE which test to run. On the DFP computer, the PTPROG.EXE file contains the diagnostic commands and executes them.

TPD\PTDIAGS\PT2\PTPROG.EXE, the DFP diagnostic test software, is run from an 18XX test program, PTDIAGS. Control of the DFP hardware is through the serial port, just as communications with DFP normally occur using PTTALK.EXE. Resulting pass/fail information is available on the Z1800-series computer.

For the DFP diagnostic tests to run successfully, all DFP hardware must be present and properly connected. The DFP computer must have between one and four Channel Control Cards installed. Each CCC is to be connected to a DR2p board installed in the Z1800 Series tester.

The goals of the DFP diagnostic tests are to verify the DFP hardware is operating as expected and to diagnose any errors. The DFP hardware tested by the DFP diagnostic tests are

- serial port connection between the computers (partially tested)
- all installed CCCs
- all installed (connected) DR2p Boards
- connections between CCCs and DR2p's

The diagnostic tests determine the condition of the CCCs and, to a limited degree, the DR2p boards, but not the condition of the DFP computer hardware.

## Tools for Custom Development

DFP offers examples of code in the following chapters of this manual and in the installed DFP data base (in the subdirectory PT2\EXAMPLES); other examples may be available from Teradyne. Copy the example file that comes closest to your needs and customize it as necessary. Verify and debug your code using PTTALK.EXE, the keyboard and monitor connected to the DFP computer, and any other instruments you may choose.

### Assemblers and Simulators

For jobs requiring DUT assembler work, Teradyne supplies the option of an assembler plus simulator packages from PseudoCorp.

**Note: If you already have an assembler for your DUT work, you can continue to use it; the PseudoCorp assembler is only necessary if you want or need to maintain or extend assembly language work that was originally written for that assembler.**

### Examples of Custom Applications

Each example of a custom application in later chapters addresses a particular mode of operation, such as serial boot or flash in free air. Each shows how its activities are coordinated with the tester, and how its activities are coordinated with activities of the DUT processor.

**Note: Other examples may be available from Teradyne, Inc.**

# Hardware—Theory of Operation 3

## Introduction

Digital Function Processor hardware consists of

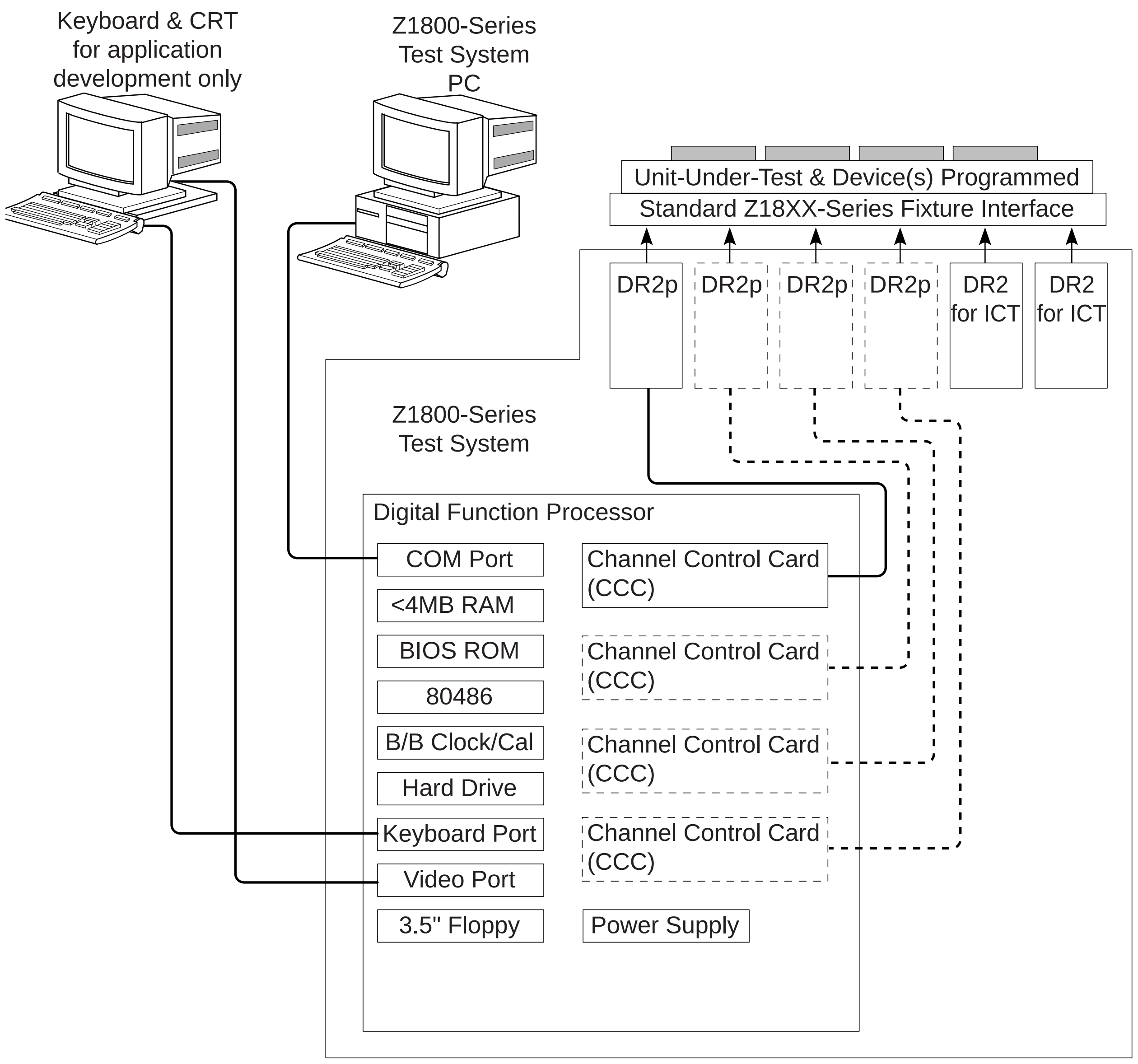
- a chassis
- a computer
- Teradyne Channel Control Cards
- a DR2p board in the Z1800-series tester

The DFP chassis contains a PC clone computer. The DFP computer has its own hard disk with 4MB of RAM, and runs MS-DOS operating system version 6.0 or later. The DFP motherboard has conventional expansion-board slots which support Teradyne Channel Control Cards (CCCs). The cable connecting DFP to the Z1800-series main computer passes through a slot which is provided for this purpose in the rear of the tabletop of the power supply bay.

DFP and the Z1800-series main computer do not operate as equals on a network. Rather, DFP is a slave to the Z1800-series main computer. The Z1800-series main computer sends commands to the DFP computer. The command interface consists of com ports connected by a null modem cable.

A DR2p board has all the in-circuit test features of the standard 32-channel DR2. In addition, DR2p channels have features that allow them to serve the needs of DFP. DR2p channels pass all standard Z1800-series self-tests and can be used as ordinary in-circuit test channels for digital Gray code, vector performance, analog, and mixed signal purposes in applications not using DFP functions. Specialization of certain DR2p channels to certain functions demands special wiring for fixtures in applications that use DFP functions.

Figure 3.1  Hardware Overview and Interconnect Diagram



Note: Digital Function Processor shown configured with
one Channel Card installed,
expandable up to four CCCs.

## DFP and Interconnections

The DFP chassis is available in various mounting configurations.

- The Z1888, Z1890, and Z1880 version mounts horizontally in the front of the console, below the test head cage.
- The Z1860 version mounts vertically in the front of the console, at the left end of the test head cage.
- The Z1840/1850 version mounts horizontally in the rear of the console.

In a Z1840, the DFP chassis mounts in the rear of the power supply bay. In a Z1850, it mounts behind the test head cage. For information about mounting in other testers, please contact Teradyne.

## Power

DFP's chassis contains a power supply
which converts the Z1800-series tester's line conditioner output voltage to the DC voltages necessary to operate the PC motherboard and its associated plug-ins. DFP is equipped with a US 3-prong 110-volt plug for connection into the plugstrip inside the tester chassis.

### Front Panel

Keyboard and CRT connectors are mounted on the front panel. While the user-supplied keyboard and CRT are not used in day-to day-testing, they are useful during application development. You can connect the keyboard and CRT by unlocking the door covering the front panel and plugging in the two connectors.

## Reset Button

The reset button is not used in normal operation. Ordinarily, reset action is caused by a relay closure commanded through the PC I/O board, using either a relay on the RAB or a relay provided for this purpose on the power supply control board. Resetting causes the DFP computer to reboot from its hard disk.

Complete hardware reset is essential to slave operation. The Z1800-series main computer can force the DFP computer to reboot from DFP's hard disk in two ways:

- A relay is provided on the power supply controller board.
- A relay is available on the RAB.

A manual push button is also mounted on the front panel of DFP.

The Z1800 system software uses ptboot.exe to reboot the DFP computer. When you install DFP software, you will choose which of the two versions of ptboot.exe to use, based on the available hardware connection.

## Floppy Disk Drive

The floppy drive in the front panel is drive A and is used for initial installation of software. You may use drive A for installing custom programs, for storing write-protected data, or for any other purpose.

## Rear

All cables to internal parts of the tester connect to the rear side of the DFP box. These cables include

- serial port cable
- AC power cable
- reset cable
- ribbon cables from DR2p boards to CCCs

## Clock

DFP has a battery-backed clock-calendar. DFP software keeps this clock in agreement with the clock in the main tester PC. Every time DFP is assigned a new task, the DFP clock automatically is set to match the main PC clock.

## Cables

Each DR2p connects to a CCC in DFP via one 80-pin flat ribbon cable. In this cable, one ground wire is supplied for every two signal wires, distributed in such a way that every signal wire is adjacent to one ground wire.

## DFP Functions

DFP includes the following applications:

- Address generator application
- Serial data application
- Parallel data application without handshake
- Parallel data application with handshake

All CCCs are physically and electrically identical except for their address switch settings. All DR2p boards are identical. Each CCC/DR2p pair assumes a function when its registers are loaded by software running in DFP.

Once a CCC is set up for one of these four functions, it operates in that capacity until you change its setup.

This document illustrates these four functions independently. It is possible to mix them to some degree, and achieve hybrid functionality, or to achieve varying functionality by changing the setup during a test.

### Address Generator Application
#### (ADDRESS mode)

In 24-bit address generator mode, the CCC provides a 24-bit address counter for addressing nonvolatile memories in free-air mode.

The counter can be cleared, loaded, and incremented by DFP's CPU.

If you expect to have DUTs with multiple address buses, you can install multiple CCCs in DFP. You can also use unused high-order address bits as disable drivers, to disable devices on the DUT during nonvolatile memory (NVM) writing.

### Serial Application
#### (SERIAL mode)

In a two-port serial application, the CCC provides two flexible serial asynchronous ports, two 8-bit parallel ports, and a 0–12.75 volt programmable voltage source and sensor.

Under program control, the serial ports can operate at a wide range of baud rates.  You can program them for word size, stop bits and parity.  The primary purpose is to bootstrap microcontrollers such as the Motorola 68HC11 and 68HC05 series. The serial port controller is a Zilog/Hitachi 85C30 chip. Teradyne recommends you obtain an 85C30 manual.

The parallel ports are simple 8-bit latches, each of which can be used for input or output. They provide reset signals at the beginning of bootstrap operations, wiggle bits during the programming of serial-access EEPROMs, and exchange single-bit messages with the DUT as needed. As outputs, these ports have the full backdrive specifications of DR2 channels.

If more than two serial ports are needed in an installation (for example, if multi-panel DUT construction is expected), you can add Channel Control Cards and program them to operate in the serial boot mode.

### Parallel Data Application Without Handshake (DATA mode)

In parallel data applications, the CCC provides two eight-bit bytewise bidirectional ports (Port A and Port B) and one eight-bit bitwise bidirectional port (Port C).Using these data ports, the DFP CPU supplies commands and data to flash memories and reads out data and status from flash memories.  Each board handles the data path to two 8-bit wide flash memory chips or to one 16-bit wide chip. Up to four CCCs can be installed along with up to four DR2p boards to serve memories on wider buses.

### Parallel Data Application With Handshake (PARALLEL mode)

Parallel Data with handshake also provides the same abilities as described above, but with handshaking capabilities. Signals originating on the DUT can act as strobes which enable the data paths from DR2p to the DUT. In addition, the CCC provides a set of signaling flipflops which allow programs running in the DUT to control the rate at which DFP provides new data to those ports.

## DFP Function and Software Control

DR2p, like DR2, has 32 channels and serves 32 nodes of the device under test (DUT). The eight lowest numbered channels (referred to here as number 0 through number 7) are associated with Port A. The next eight are associated with Port B, and the third group, Port C. The fourth group, numbered 24 to 31, have special purposes, are not associated with a specific port, and generally do not have logic-level signals on them.

Ports A and B are similar to each other in concept, working bytewise. Port C is controlled bitwise.

Figure 3.2 DFP's DR2p Resources

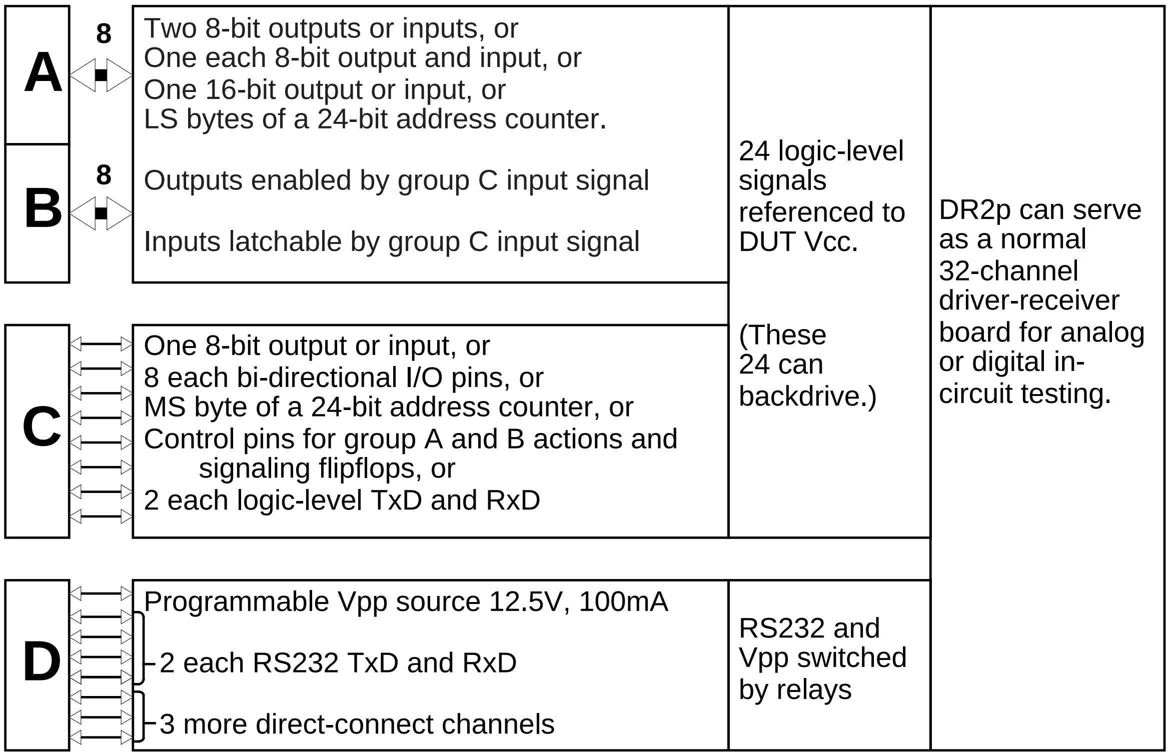

A
8
Two 8-bit outputs or inputs, or
One each 8-bit output and input, or
One 16-bit output or input, or
LS bytes of a 24-bit address counter.
B
8
Outputs enabled by group C input signal
Inputs latchable by group C input signal
24 logic-level signals referenced to DUT Vcc.
(These 24 can backdrive.)
DR2p can serve as a normal 32-channel driver-receiver board for analog or digital in-circuit testing.
C
One 8-bit output or input, or
8 each bi-directional I/O pins, or
MS byte of a 24-bit address counter, or
Control pins for group A and B actions and signaling flipflops, or
2 each logic-level TxD and RxD
D
Programmable Vpp source 12.5V, 100mA
2 each RS232 TxD and RxD
3 more direct-connect channels
RS232 and Vpp switched by relays

**Table 3.1 Channel Functionality in Various Applications**

| Ch. | Functional Mode | | | |
|---|---|---|---|---|
| | **PARALLEL (Parallel w/ Handshake)** | **ADDRESS (Address)** | **SERIAL (Serial)** | **DATA (Parallel no Handshake)** |
| 0 | PortA D0 | A0 (o) | PortA D0 | PortA D0 (io) |
| 1 | PortA D1 | A1 (o) | PortA D1 | PortA D1 (io) |
| 2 | PortA D2 | A2 (o) | PortA D2 | PortA D2 (io) |
| 3 | PortA D3 | A3 (o) | PortA D3 | PortA D3 (io) |

| Ch. | Functional Mode | | | |
|---|---|---|---|---|
| | PARALLEL (Parallel w/ Handshake) | ADDRESS (Address) | SERIAL (Serial) | DATA (Parallel no Handshake) |
| 4 | PortA D4 | A4 (o) | PortA D4 | PortA D4 (io) |
| 5 | PortA D5 | A5 (o) | PortA D5 | PortA D5 (io) |
| 6 | PortA D6 | A6 (o) | PortA D6 | PortA D6 (io) |
| 7 | PortA D7 | A7 (o) | PortA D7 | PortA D7 (io) |
| 8 | PortB D0 | A8 (o) | PortB D0 | PortB D0 (io) |
| 9 | PortB D1 | A9 (o) | PortB D1 | PortB D1 (io) |
| 10 | PortB D2 | A10 (o) | PortB D2 | PortB D2 (io) |
| 11 | PortB D3 | A11 (o) | PortB D3 | PortB D3 (io) |
| 12 | PortB D4 | A12 (o) | PortB D4 | PortB D4 (io) |
| 13 | PortB D5 | A13 (o) | PortB D5 | PortB D5 (io) |
| 14 | PortB D6 | A14 (o) | PortB D6 | PortB D6 (io) |
| 15 | PortB D7 | A15 (o) | PortB D7 | PortB D7 (io) |
| 16 | Data RdyA (o) | A16 (o) | TxdA (o) | PortC D0 (io) |
| 17 | Data EnA (i) | A17 (o) | RxdA (i) | PortC D1 (io) |
| 18 | Data StbA1 (i) | A18 (o) | PortC D2 (io) | PortC D2 (io) |
| 19 | TestA (i) | A19 (o) | PortC D3 (io) | PortC D3 (io) |
| 20 | Data RdyB (o) | A20 (o) | TxdB (o) | PortC D4 (io) |
| 21 | Data EnB (i) | A21 (o) | RxdB (i) | PortC D5 (io) |
| 22 | Data StbB1 (i) | A22 (o) | PortC D6 (io) | PortC D6 (io) |
| 23 | TestB (i) | A23 (o) | PortC D7 (io) | PortC D7 (io) |
| 24 | Vpp (o) | Vpp (o) | Vpp (o) | Vpp (o) |
| 25 | Ch25 | Ch25 | RS232 TXD A *1 | Ch25 |
| 26 | Ch26 | Ch26 | RS232 RXD A *1 | Ch26 |
| 27 | Ch27 | Ch27 | RS232 TXD B *1 | Ch27 |
| 28 | Ch28 | Ch28 | RS232 RXD B *1 | Ch28 |
| 29 | Ch29 | Ch29 | Ch29 | Ch29 |
| 30 | Ch30 | Ch30 | Ch30 | Ch30 |
| 31 | Ch31 | Ch31 | Ch31 | Ch31 |

**Note: Port A and Port B are bytewise controlled; Port C is bitwise controlled.**

## Addresses on CCCs

The base address of a CCC is established by an array of DIP switches. Once the base address has been set, registers on the CCC will have defined addresses. Your application program will write to or read from these addresses to control the modes of operation and to exchange information with the device under test.

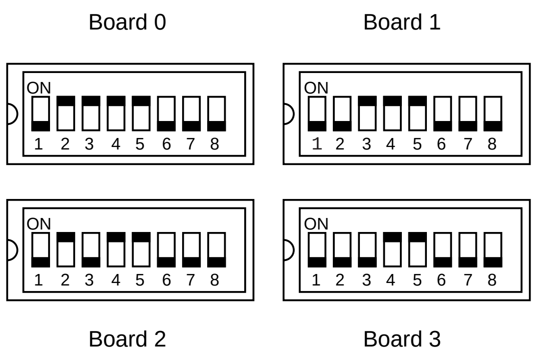
**Table 3.2 Base Addresses for CCCs**

| | |
|---|---|
| CARD 0 | 0x0380 |
| CARD 1 | 0x0388 |
| CARD 2 | 0x0390 |
| CARD 3 | 0x0398 |

The specific address of a control register or data port is a combination of the case CCC address with a “Block Number” and an “offset.” You will ordinarily use library mnemonics instead of numbers for register addresses to make your programs more readable. Both methods are presented in the following register descriptions.

**Note: Refer to your DFP installation guide or to the section “CCC Address Management” later in this chapter for a discussion of these switches and address generation.**

Figure 3.3 DIP Switch Settings



**Note: SW1, position 1 must always be off.**

**Table 3.3 Programmer’s IO Port Map**

| Block | Offset | Mode | Description |
|---|---|---|---|
| 0 | 0 | W16 | Prime address 0–15 register |
| 0 | 2 | W8/R16 | Load address 16–23 and update register |
| 0 | 4 | W16 | Load address 0–15 and update register |
| 0 | 6 | W8 | Inc/Dec address register |
| 1 | 0 | W8 | Port C control |
| 1 | 2 | W16 | Port A/B control and Port D aux. relays |
| 1 | 4 | W8 | Reset Port A handshake |
| 1 | 6 | W8 | Reset Port B handshake |
| 2 | 0 | W16 | Gate Array instruction |
| 2 | 2 | W16 | Gate Array Enables |

| Block | Offset | Mode | Description |
|---|---|---|---|
| 2 | 4 | W16 | Gate Array Data |
| 2 | 6 | W16 | Gate Array Gray Code |
| 3 | 0 | RW 8 | Port A data |
| 3 | 2 | RW 16 | Port B data—data on bits 8-15 |
| 3 | 4 | W8 R16 | Port C data |
| 3 | 6 | RW 16 | Port A+B data |
| 4 | 0 | RW 16 | CTC Counter/Timer 0 |
| 4 | 2 | RW 16 | CTC Counter/Timer 1 |
| 4 | 4 | RW 16 | CTC Counter/Timer 2 |
| 4 | 6 | W8 | CTC Mode Register |
| 5 | 0 | RW 8 | SIO Ch B command |
| 5 | 2 | RW 8 | SIO Ch B data |
| 5 | 4 | RW 8 | SIO Ch A command |
| 5 | 6 | RW 8 | SIO Ch A data |
| 6 | 0 | R8 | Card Status |
| 6 | 2 | W8 | Software Reset |
| 6 | 4 | W8 | Vpp Data Register |
| 6 | 6 | W8 | Misc. Register |
| 7 | 0 | W8 | Trigger timer 0 |
| 7 | 2 | W8 | Trigger timer 1 |
| 7 | 4 | W8 | Trigger timer 2 |
| 7 | 6 |  | Not used |

| Legend: |  |  |
|---|---|---|
|  | **W** | **Write** |
|  | **R** | **Read** |
|  | **8** | **8 bits wide** |
|  | **16** | **16 bits wide** |

**Note: Take care to use the proper C-language software with an 8-bit wide installation or 16-bit wide installation; writing or reading with the wrong word size causes unpredictable operation, but no error messages are produced.**

## Register Descriptions

The I/O registers are described below (also see Table 3.3). Each description includes examples: first an example using the straight addressing format, then the same example using the pt2.lib define statements or function calls. The define statements and function calls should make programming easier for the user.

**Note: The total list of define statements and function call descriptions available via pt2.h and pt2.lib can be found in p2t.h (see the chapter including the listing of pt2.h).**

### Block 0—Address Counter

**Offset 0.** Store a 16-bit data value in the Least Significant (LS) 16 bits of the address counter. The actual output register which drives the DUT pins is not updated.

**Offset 2.** Store an 8-bit value in the upper byte of the address counter, then update the output register with all 24 bits of the counter in parallel.

**Offset 4.** Store a 16-bit value in the LS16 bits of the address counter, then update the output register with all 24 bits of the counter in parallel.

**Offset 6.** Increment or decrement the address counter, according to the miscellaneous register direction control, then update the output register with all 24 bits of the counter in parallel.

Example without define statements:

```
      0000       Block
     +0380       Card Address (CCC 0)
     +   6       Offset
outp(0x0386,0x00); //increment/decrement register
                  //(0x00 dummy data)
```

Example with pt2.lib define statements:

```
includes functions:
      setCountDir (refer Block 6, offset 6)
      setCount    (refer Block 0, offset 0)
      incCount    (refer Block 0, offset 6)

setCountDir(0,UP);                 //set card 0 to count up
setCount(0,0x03, 0x0000);          //DUT address = 0x030000
incCount(0);                       //increment address by 1
```

## Block 1—Port Control

**Offset 0 - Port C**

Write Port C control information derives from whether a bit should be an input or an output and is determined by the DUT and CCC functionality.

```
bit definition:      0 = input (to DFP)
                     1 = output (from DFP)
```

**Note:** **Additional function control for Address/Serial/Parallel modes are found under Block 6, offset 6.**

Example initialization values for Port C (refer also to Table 3.2):

```
1 - Address mode with 19 address pins  :     0x07
    (AO-15 via Port A+B, A16-18 via port C)
         bit 0 - output -> DUT address bit 16
         bit 1 - output -> DUT address bit 17
         bit 2 - output -> DUT address bit 18

2 - Serial mode - 1 device             :     0x01
         CCC TxdA - serial output -> DUT serial Rxd
         CCC RxdA - serial input <- DUT serial Txd

3 - Serial mode - 2 devices            :     0x11
         CCC TxdA - serial output -> DUT A serial Rxd
         CCC RxdA - serial input <- DUT A serial Txd
         CCC TxdB - serial output -> DUT B serial Rxd
         CCC RxdB - serial input <- DUT B serial Txd

4 - Parallel mode without HS           :     0x07
         (also called data mode)
         bit 0 - output -> DUT WE*
         bit 1 - output -> DUT OE*|
         bit 2 - output -> DUT CE*

5 - Parallel mode without HS           :     0x1D
         (also called data mode)
         bit 0 - output -> DUT CE*
         bit 1 - output -> DUT WE*
         bit 2 - output <- DUT serial out (SO)
         bit 3 - output -> DUT serial in (SI)
         bit 4 - output -> DUT clock

6 - Parallel data mode with HS - 2 devices   :  0x11
         Data RdyA - output ->   DUT
         Data EnA  - input <-    DUT
         Data RdyA - input <-    DUT

         Data RdyB - output ->   DUT
         Data EnB  - input <-    DUT
         Data RdyB - input <-    DUT
```

Example - Port C output for address mode -
19 address pins (A16-19 via Port C):

A. Without define statements:

```
    1000       Block
   +0388       Card Address (CCC 1)
   +   0       Offset
outp(0x1388, 0x07);  //C port, bits 0-2 output
```

B. With pt2.lib define statements:

```
outp(ca[1]|PC_CNTL, 0x07);  //C port, bits 0-2 output
```

or

```
outp(CARD1|PC_CNTL, 0x07);  //C port, bits 0-2 output
```

Example - Port C output for serial mode using Channel A:

A. Without define statements:

```
       1000       Block
      +0388       Card Address (CCC 1)
      +   0       Offset
outp(0x1388, 0x01);  //C port, bit 0 output= CCC TXDA
                     //     bit 1 input = CCC RXDA
```

B. With pt2.lib define statements:

```
   includes : PC_CNTL
              PC_DATA      (refer Block 3, offset 4)

outp(ca[1]|PC_CNTL, 0x01); //C port, bit 0 out = CCC TXDA
                           //        bit 1 in  = CCC RXDA
outp(ca[1]|PC_DATA, 0x00); //setup DR2P U73 to for serial
                           //mode
```

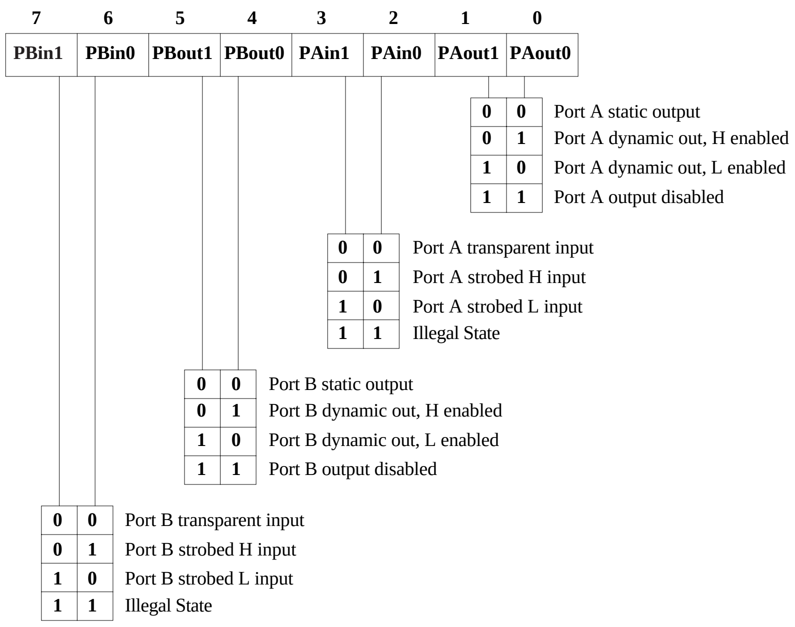
or

```
outp(CARD1|PC_CNTL, 0x01); //C port, bit 0 out = CCC TXDA
                           //        bit 1 in  = CCC RXDA
outp(CARD1|PC_DATA, 0x00); //setup DR2P U73 to for serial
                           //mode
```

**Offset 2. Write Port A,B,D control information.**

**Port A & B:** The control information indicates whether a bit should be an input (transparent or strobed) or an output (static or dynamic), which is determined by their DUT and CCC functionality. The lower 8 bits (bits 0-7) determine how Port A&B function, as shown.

**Table 3.4 Port A & B Control**



| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| PBin1 | PBin0 | PBout1 | PBout0 | PAin1 | PAin0 | PAout1 | PAout0 |

| PAout1 | PAout0 | |
|---|---|---|
| 0 | 0 | Port A static output |
| 0 | 1 | Port A dynamic out, H enabled |
| 1 | 0 | Port A dynamic out, L enabled |
| 1 | 1 | Port A output disabled |

| PAin1 | PAin0 | |
|---|---|---|
| 0 | 0 | Port A transparent input |
| 0 | 1 | Port A strobed H input |
| 1 | 0 | Port A strobed L input |
| 1 | 1 | Illegal State |

| PBout1 | PBout0 | |
|---|---|---|
| 0 | 0 | Port B static output |
| 0 | 1 | Port B dynamic out, H enabled |
| 1 | 0 | Port B dynamic out, L enabled |
| 1 | 1 | Port B output disabled |

| PBin1 | PBin0 | |
|---|---|---|
| 0 | 0 | Port B transparent input |
| 0 | 1 | Port B strobed H input |
| 1 | 0 | Port B strobed L input |
| 1 | 1 | Illegal State |

"Static output operation" means that the port is always active and does not float.

When a port is in dynamic output mode, a channel is used to enable and tristate the port. Channel 17 is used for port A and channel 21 for port B—I17 and I21 in the DR2p U74 PAL equations. This signal usually is generated by the DUT.

**Port D:** The control information indicates whether the DR2p relay is open or closed. These are the direct relays that can handle the VPP and RS232 voltages. The upper 8 bits (bits 8-15) determine whether the D port relays are opened or closed, each bit representing the corresponding channel (bit 8= channel 24, bit 9= channel 25, etc.)

```
bit definition: 0 = closed
                1 = open
```

Code Example  Port A = static output,   transparent input
              Port B =  output disabled, transparent input
              Port D = channel 0 (VPP) DR2P:relay closed(set)
                       channel 1-7 DR2P    :relays open  (reset)

**A. Without define statements:**

```
             1000          Block
            +0380          Card Address (CCC 0)
            +   2          Offset
   outpw(0x3382,0xFE30);       //A out, B in, VPP set
```

FE30:

```
        FE = 1111 1110 ->bit 0 = VPP DR2P relay set
        30 = 0011 0000 ->bit 0,1 = PAout= static out
                         bit 2,3 = PAin = xparent in
                         bit 4,5  = PBout = out dis-
abled
                         bit 6,7 = PBin  = xparent in
```

(See  Table 3.4)

**B. With pt2.lib define statements:**

```
outpw(ca[0]|PAB_CNTL, 0xFE30);  //A out, B in, VPP set
```

**Note: Further control for Address/Serial/Parallel modes is explained in the section covering Block 6, Offset 6 later in this chapter.**

**CAUTION** **Since the D port auxiliary relays are often used with voltage levels above or below the TTL levels normally used by the DR2P (VPP, RS232), these relays MUST be cleared (opened) before the DFP returns control of the DR2P card/s to the 18xx computer. Failing to clear these relays may result in damage to the tester. The easiest way to clear these, as well as all DR2P relays used during the DFP program, is with the pt2.lib function- release DR:**

```
        releaseDR(0);   //clear all relays - card 0.
        releaseDR(1);   //clear all relays - card 1.
                        //refer pt2.h
```

**Offset 4.** Write to this address to reset the Port A handshake logic. A hard reset also achieves this effect.

**Offset 6.** Write to this address to reset the Port B handshake logic. A hard reset also achieves this effect.

## Block 2—Gate Array

This block of addresses writes to the DR2p gate arrays, primarily to allow control of their relays, although all other functions can be controlled. The bit values corresponding to tester backplane data bits are tabulated below. (Refer to DR2P schematic)

All data writes are 16 bit, and convey data to the gate array coded as shown below.

B0 = Inst 0 or Data 0
B1 = Inst 1 or Data 1
B2 = Inst 2 or Data 2
B3 = Inst 3 or Data 3

B4 = Inst 4 or Data 4
B5 = *All Nodes or Data 5
B6 = Node addr(0) or Data 6
B7 = Node addr(1) or Data 7

B8 = Inst(5)
B9 = Inst(6)
B10 = Inst(7)
B11 = Card select

B12 = Node addr(2)
B13 = Node addr(3)
B14 = Node addr(4)
B15 = Not used

**Note: The actual coding needed for writing to the gate arrays can be found via the define statements in the pt2.h file; see the chapter covering pt2.h.**

**Offset 0**. Gate array Instruction. Generates a 'INST_STROBE' signal to place an instruction in the gate arrays. Used to set D, E, F, and G relays.

**Offset 2.** Gate array Enables. Generates a 'E_STEP_Ck' signal to clock the gate arrays. Currently not used in DFP operations.

**Offset 4.** Gate array Data. Generates a 'D_STEP_Ck' signal to clock the gate arrays. Currently not used in DFP operations.

**Offset 6.** Gate array Gray Code Data. Generates a 'Data_STROBE' signal to latch data into the gate arrays. Currently not used in DFP operations.

Example - Set DR2P D-reed channel 0, card 0:

A. Without define statements:

```
      2000         Block
     +0380         Card Address (CCC 0)
     +   0         Offset
outpw(0x2388, 0x29);    //Set D-reed position 0
```

```
Note: 0x29: = 0x20 - node address 0 (channel 0)
             +0x08 - D-reed
             +0x01 - Set
```

**(See the pt2.h define statements)**

B. With pt2.lib define statements:

```
   outpw(ca[0]|GA_INST,D_REED|nodA[0]|SET);
or
   outpw(CARD0|GA_INST,D_REED|nodA[0]|SET);
```

## Block 3—Port A/B/C Data

**Offset 0.** Write to set Port A data. Depending on the mode selected for this port, data may be enabled to DUT or may be held in high Z mode. Read to return value of Port A data. Data returned will be last data latched, if port is set into latched mode.

**Offset 2.** Write to set Port B data. Depending on the mode selected for this port, data may be enabled to DUT or may be held in high Z mode. Read to return value of Port B data. Data returned will be last data latched, if port is set into latched mode. Note that Read/Write Port B is a 16-bit operation, with Port B data passed via the upper 8 bits (bits 8 through 15).

**Offset 4.** Write to set Port C data in either parallel or serial mode. In address generator mode, port C will be updated as part of the counter programming in block 0. If a 16-bit address counter is used, Port C is free to be used for other functions and can then be read or written. Note that Read Port C is a 16-bit operation with Port C data bits 0 through 3 returned via bits 0 through 3 and Port C data bits 4 through 7 returned via bits 8 through 11.

**Note: If using either serial or parallel with handshake modes, it is necessary to perform a preliminary port data write of 0x00 following the C port control write. This will enable the ARDY/BRDY, TXDA/TXDB signals via U73 on the DR2P card (See the U73 PAL Equations in this chapter).**

Example - Read Port C data / Write Port C data:

A. Without define statements for Read Port C:

```
      3000          Block
     +0390          Card Address (CCC 2)
     +   4          Offset
   inpw(0x3394);    //Read Port C data
```

```
Note: read port A = returned: bits 0-7  = data A 0-7
      read port B = returned: bits 8-15 = data B 0-7 !!!
      read port C = returned: bits 0-3  = data C 0-3
                              bits 8-11 = data C 4-7 !!!
```

(See Table 3.3)

**Note:** **C software functions:**

```
inp/outp   = 8  bit read or write
inpw/outpw = 16 bit read or write
```

B. With pt2.lib define statements for Read and Write Port C:

```
        includes: PC_CNTL - Port C input
                  PC_DATA - Read data Port C
                  PC_CNTL - Port C output
                  PC_DATA - Write data Port C

   outp(ca[2]|PC_CNTL,0x00);     //Port C in
   data = inpw(ca[2]|PC_DATA);   //data read
   outp(ca[2]|PC_CNTL,0xFF);     //Port C out
   outp(ca[2]|PC_DATA,0x55);     //data write 55

or

   outp(CARD2|PC_CNTL,0x00);     //Port C in
   data = inpw(ca[2]|PC_DATA);   //data read
   outp(CARD2|PC_CNTL,0xFF);     //Port C out
   outp(CARD2|PC_DATA,0x55);     //data write 55
```

**Offset 6.** Write/Read Port A and Port B data together to allow concatenation of Port A and Port B into a 16-bit parallel port.

## Block 4—CTC

The DFP interface board (PN 051-003-00) provides three timers via an 82C53 package, allowing the construction of accurate timing loops. The timers have 500 nanosecond resolution and 10 millisecond maximum time and can be polled.Teradyne recommends DFP programmers obtain the manual for the 82C53 from Intel.

The 82C53 registers are available as follows:

**Offset 0.** 82C53 Counter/Timer 0.

**Offset 2.** 82C53 Counter/Timer 1.

**Offset 4.** 82C53 Counter/Timer 2.

**Offset 6.** 82C53 Mode register 3.

The counter is configured by on-board logic to be used in mode 1. This gives three timers, each with retriggerable one-shot functionality. The timing function is triggered by writes to addresses in block 7.

The timeout status of each channel can be read as part of the board status register—block 6 offset 0.

Bit 2 = counter 0
Bit 3 = counter 1
Bit 4 = counter 2

The data bits will be returned as a logic 1 when timeout is complete for the respective channel.

For further information on programming timers, obtain an 8253 data manual from Intel or another 8253 manufacturer.

Example - Set Counter/Timer 1 to mode 1:

A. Without define statements:

```
        4000          Block
       +0380          Card Address (CCC 0)
       +   6          Offset
   outp(0x4386,0x32);    //Set timer 1 to mode 1
```

(See manufacturer's manual for 8253 chip).

B. With pt2.lib function:

```
   initCard(0);      //inits card 0 -includes
                     //set all three timers to mode
```

Below is an example of how you may use the CTC timer using PT2.LIB defines and functions:

```
includes: initCard()- timer in mode 1
          CTC_CNTL0 - timer 0 count
          TRG_0     - trigger timer
                          (Refer Block 7)
         readStatus - read misc. status
                          (refer Block 6, Offset 0)
```

```
   initCard(0);                         //set timers to mode = 1
   outp(ca[0]|CTC_CNTL0,(unsigned int) 10);//set timer 0 for count = 10
   outp(ca[0]|TRG_0,0x00);              //trigger timer 0 to start count
                                        // (0x00 dummy data)
   if((readStatus(0)) & 0x04);     //check status = set
or
   while(!(readStatus(0) & 0x04)); //wait status = set
                                   //'& 0x04' masks for bit 2=timer 0 status
```

**Note: The timing may vary slightly between DFP computers depending on the BIOS setup and compiler/options.**

## Block 5—SIO

The DFP interface board (PN 051-003-00) provides an 85C30 SIO chip to allow the transmission/reception of asynchronous serial data with flexible selection of baud rates.

The 85C30 registers are available as follows:

**Offset 0.** 85C30 channel B command register.

**Offset 2.** 85C30 channel B data register.

**Offset 4.** 85C30 channel A command register.

**Offset 6.** 85C30 channel A data register.

Example - Select register 9 of Channel A of the 85C30:

A. Without define statements:

```
      5000          Block
     +0380          Card Address (CCC 0)
     +   4          Offset
 outp(0x5384,0x09);   //select 85C30 CH A, reg 9
```

B. With pt2.lib define statements:

```
        includes: SIO_ACOM - select command register
                  SIO_DAT  - data to write to register

 outp(ca[0]|SIO_ACOM,0x9);  //select 85C30 CH A, reg 9
 outp(ca[0]|SIO_ADAT,0xc0); //write 85C30 reset
```

C. With pt2.lib function for SIO COM write:

```
    includes:  sioComWr - select reg and write data

sioComWr(ca[0],SIO_ACOM, 9, 0xc0);//85C30, CH A, reset
```

For further information on programming the 85C30, obtain an 85C30 data manual from Zilog or Hitachi, and look at the custom examples in this manual.

## Block 6—Miscellaneous Controls

The voltage source provides Vpp to the DUT (usually 12 volts +/- 5%), or supplies the 9-volt signal needed to initiate the bootstrap process on 68HC05 microcontrollers.

The voltage sensor detects Vpp if it is generated by circuitry on the DUT. The easiest way to use this feature is to use the pt2.lib function measVpp(); the A-D circuitry measures the voltage by successive approximation, returning an integer value which needs to be multiplied by .05 for the actual voltage (refer to the pt2.h listing in Chapter 7).

**Offset 0.** Read board status. Bit values returned are:

B0 = Channel A handshake flag bit.
B1 = Channel B handshake flag bit.
B2 = Timer 0 status.
B3 = Timer 1 status.
B4 = Timer 2 status.
B5 = Power supply overload status.
B6 = Address counter Up/Down control bit.
B7 = Address counter Up/Down control bit.

Bits 6 and 7 are designed to be used for self tests.

Example 1 - Check for channel A handshake bit.

A. Without define statements:

```
              6000       Block
             +0380       Card Address (CCC 0)
             +   0       Offset
if(inp((0x6380) & 0x01));  //check CH A handshake bit
                            //'& 0x01' = mask for bit 0
```

B. With pt2.lib define statements:

```
if(inp((ca[0]|STATUS) & 0x01)); //check CH A handshake bit
                                //'& 0x01' = mask for bit 0
```

Example 2 - Measure VPP supplied by DUT:

A. With pt2.lib function call measVpp():

```
   value = measVpp(0);   //returns integer value
   volts = value * .05;  //value * .05 = actual voltage
```

**Note: measVpp returns approximated integer value via successive readings of the misc. status register, bit 5 - the power supply overload status.**

**Offset 2.** Perform software reset on control logic. To provide full initialization, all port control registers must be updated.

Example - Reset card 3:

```
A. Without define statements:

         6000          Block
        +0398          Card Address (CCC 3)
        +   2          Offset
   outp(0x638A,0x00);   //reset card 3
                        //(0x00 dummy data)
```

B. With pt2.lib define statements:

```
   outp(card[0]|RST,0x00);   //reset card 3
                            //(0x00 dummy data)
```

C. With pt2.lib function initCard:

```
   initCard(3); //inits card 3 -includes
                //software reset.
```

(See pt2.h chapter.)

**Offset 4.** `Write Vpp supply voltage to DAC. Vpp is programmable from 0 to 12.75v in 0.05v increments. Vpp must be enabled by the miscellaneous register (block 6, offset 6, bit 1) and by the DR2p board relay (block 1, offset 2, bit 8).`

Example - Set card 0 Vpp to 12v:

A. Without define statements:

```
          6000          Block
         +0380          Card Address (CCC 0)
         +   4          Offset
   outp(0x6384,0xF0);   //setup Vpp Card 0 to 12V
```

B. With pt2.lib define statements:

```
   outp(card[0]|VPP,0xF0);  //set Vpp Card 0 to 12V
or
   outp(CARD0|VPP,0xF0);  //set Vpp Card 0 to 12V
```

C. With pt2.lib defines and function:

```
 includes: PAB_CNTL - set D-reed for VPP (Block 1,offset 2)
           setVPP   - set VPP voltage    (Block 6,offset 4)
           VPP          - VPP on         (Block 6,offset 6)
           delay        - 50 ms delay
           VPP          - VPP off        (Block 6,offset 6)

outpw(ca[0]|PAB_CNTL,0xFE00); //VPP set, Port A,B = out
setVpp(0,240);                //VPP = 12V(240 x .05 = 12)
selVpp(0,ON);                 //turn on Vpp
delay(50);                    //50ms wait for VPP
selVpp(0,OFF);                //turn off Vpp
```

**Offset 6.** Write to Miscellaneous Register. This register contains the odd control bits used by the control board. They are defined as follows:

B0 = Set to 1 for serial function and 0 for address or data.
B1 = Vpp on/off. 1 = on, 0 = off.
B2 = Address counter up/down. 1 = up, 0 = down.
B3 = Enable RS232 driver. 1 = enable, 0 = disable.

B4 = Dual purpose:
Select serial mode: TTL levels or RS232 levels
0 = TTL, 1 = RS232;
Selects handshake:
0 = handshake, 1 = no handshake (Port C = I/O)

B5 = Enable DFP mode.
0 = normal DR2 functions, 1 = DFP functions.

B6 = Invert Tx/Rx data Channel A. 0 = normal, 1 = invert.

B7 = Invert Tx/Rx data Channel B. 0 = normal, 1 = invert.

Example - Card 0 = Address mode

count up (addresses)

enable dfp/pt2

A. Without define statements:

```
         6000           Block
        +0380           Card Address (CCC 0)
        +   6           Offset
    outp(0x6386,0x24);

Note:  0x24: 0010 0100 -> bit 0 = 0 = address mode
                          bit 1 = 0 = Vpp off
                          bit 2 = 1 = count up
                          bit 3 = 0 = disable RS232 driver
                          bit 4 = 0 = if serial mode -
                                               TTL level
                          bit 5 = 1 = Enable DFP
                          bit 6 = 0 = if serial mode -
                                           Ch A normal mode
                          bit 7 = 0 = if serial mode -
                                            Ch B normal mode
```

(See Table 3.3)

B. With pt2.lib define statements:

```
    outp(ca[0]|MISC,0x24);
or
    outp(CARD0|MISC,0x24);
```

C. With pt2.lib functions -

```
setMode(0,ADDRESS); //address mode, card 0
ptEnable(0,ON);     //enable DFP, card 0
setCountDir(0,UP);  //direction of count is up, card 0
```

**Note: There are four pt2.lib defined modes for the setMode function (See Table 3.2 and the pt2.h chapter):**

```
ADDRESS  = address mode
SERIAL   = TTL level serial mode
DATA     =  parallel mode without handshake
PARALLEL = parallel mode with handshake
```

## Block 7—Timer Trigger

**Offset 0.** Write of any data to this address starts timer 0.

**Offset 2.** Write of any data to this address starts timer 1.

**Offset 4.** Write of any data to this address starts timer 2.

**Offset 6.** Not currently used.

Example  - Start counter/timer 0:

A. Without define statements:

```
          7000          Block
         +0380          Card Address (CCC 0)
         +   0          Offset
   outp(0x6380,0x00);   //trigger timer 0
                      //(0x00 dummy data)
```

B. With pt2.lib define statements:

```
outp(ca[0]|TRG_0,0x00); //trigger timer 0 to start count
                        // (0x00 dummy data)
```

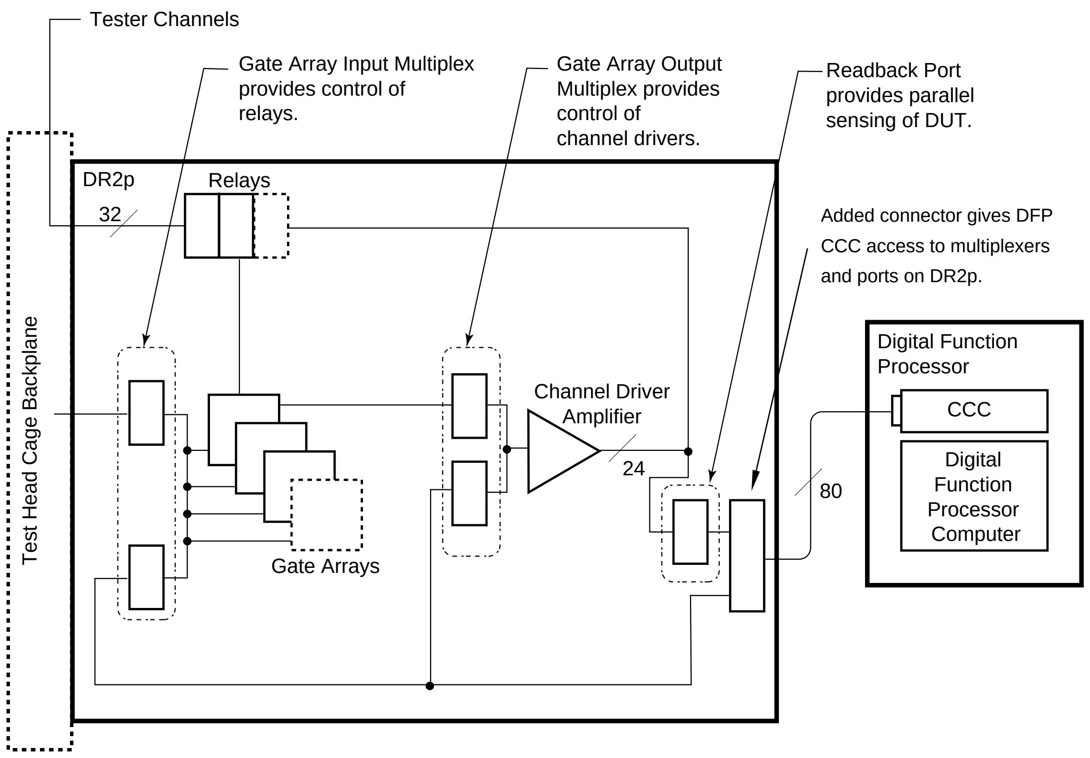
## Logic Design—DR2p

DR2p has all the functions of a DR2 and passes all DR2-oriented self test diagnostics. Thus, an application that does not use DFP can be run on a tester equipped with DFP without need to consider where the DFP channels are located in the fixture receiver. The DR2p channel locations are important only to applications that use the features of DFP.

DFP installations may contain from one to four DR2p boards. If DR2p boards are present in a tester, they are usually placed in the top four slots of the 320- node “small fixture” zone so that applications using small fixtures can reach them. Therefore the first DR2p will be in position 6 (the seventh channel board in the test head cage), and will serve nodes 192 through 223. The second will be in position 7 (the eighth channel board in the test head cage) and will serve nodes 224 through 255, and so forth.

DR2p schematics, found in the **Engineering Reference Drawings** manual for each specific Z1800-Series tester,  are helpful in the following discussion. Examine and compare them to DR2 schematics to see what has been added to DR2 to make DR2p.

Figure 3.4 Block Diagram, DR2p Board


Tester Channels
Gate Array Input Multiplex provides control of relays.
Gate Array Output Multiplex provides control of channel drivers.
Readback Port provides parallel sensing of DUT.
DR2p
Relays
32
Added connector gives DFP CCC access to multiplexers and ports on DR2p.
Test Head Cage Backplane
Digital Function Processor
CCC
Digital Function Processor Computer
Channel Driver Amplifier
24
80
Gate Arrays

The Block Diagram, Figure 3.3, illustrates DFP's general design. It does not contain all the information, nor does it show to which channels the multiplexing applies.

The block labeled Gate Array Input Multiplex allows the DFP computer to talk directly to the inputs of the gate arrays. This allows DFP to control the E, F, G and D reed relays. DFP does not use the gate arrays to apply digital signals to the DUT.

The block labeled Gate Array Output Multiplex allows the DFP computer to talk directly to the driver amplifiers. Thus DFP can stimulate the DUT directly, regardless of the state of the gate arrays, and neither depend on nor interfere with the Test Head Controller or VP.

The block labeled Readback Port allows the DFP computer to receive parallel arrays of bits from the DUT.

## Gate Array Input Multiplex

Inputs from the Z18xx backplane, which originate in the Test Head Controller (or Vector Processor Card, if your system is VP-equipped), are enabled by the pin 19 inputs of the four mux chips U67, U75, U66 and U71. The DFP inputs come via the LS16 of the GBUS. The pin 1 inputs enable this path.

The pin 1 signal is labeled EPT* which is an abbreviation for Enable PROMPTest. The pin 19 signal is labeled ENORM* which is an abbreviation for Enable NORMal. These two signals are mutual complements.

**Note: Changing a DR2p from PROMPTest DFP mode to NORMal mode, or back again, does not bring about any automatic hardware initialization. Therefore damage could result if an application does not clean up after itself. It is the programmer's responsibility to ensure safe operation before, during, and after use of the DR2p in DFP mode.**

## The GBUS

The GBUS itself is a 24-bit bus used in DFP functions. There is no equivalent to the GBUS in the ordinary DR2 board. The GBUS communicates with the CCC via the 80-pin ribbon cable. The CCC can send information to the GBUS or receive information from it. Connector J4 receives the 80-conductor ribbon cable whose other end is connected to a CCC.

The cable is terminated with capacitively coupled 82 ohm resistors. The GBUS itself is pulled up with 22k resistors.

Devices U64, U68 and U72 are bidirectional buffers, enabled by signal GBUSENABLE*. GBUSIN controls their direction. GBUSIN is high when the CCC is driving the GBUS, and low when the CCC is taking data from the GBUS. As noted earlier, some functions involve 8-bit transfers, some involve 16-bit transfers, and take place at different positions on this bus. All 24 bits of the GBUS are switched for every operation, regardless of the width or position of the word being transferred.

## Other Signals on J4

Several other signals that connect the CCC with DR2p are treated individually rather than as members of a bus.

DIR, ARDY, BRDY, A0, A1, 1E are covered in the discussion of U73—the Address Decode PAL.

ESY* and DSY* are available to combine with the normal ESYNC and DSYNC signals if necessary. In practice these two are rarely used.

ISTB* and DSTB* similarly combine with INST STROBE* and DATA STROBE* to load registers in the gate arrays. INST STROBE* is used in controlling the E, F, G and D reed relays.

EPT* and ENORM* are complements, as noted earlier, and are used to enable PROMPTest functions in opposition to NORMAL functions.

I21 and I17 return logic-level serial bitstreams to the CCC’s two serial ports.

EPT8–15 (ENABLE PROMPTest 8–15) returns a handshake signal to the CCC. In the handshake mode, whenever the DUT strobes in data from Port B, a flipflop in the CCC records the fact and allows the software to make the next data element available. This signal notifies the CCC of the event, although the actual strobed enabling of data onto the DUT’s data bus takes place on the DR2p.

EPT0–7 works just like EPT8–15, except with respect to Port A instead of B.  Ports A and B have separate signaling flipflops, so one CCC/DR2p pair can effectively support two 8-bit DUT data buses.

The eight DIRECT connections correspond to channels 24 through 31 of the DR2p board and are used to conduct analog or other non-logic-level signals directly between the CCC and the DUT. For example, the Vpp is assigned to channel 24. The CCC also provides RS232 levels in serial mode, and these DIRECT connections are used to carry the RS232 signals.  Other uses are reserved for future versions of the CCC.

The CCC can be equipped with a custom daughterboard that can communicate directly with the DIRECT signals. This is discussed in the section titled Channel Control Cards later in this chapter.

All eight DIRECT signals are isolated by relays from the channels they serve. The isolation relays are on the DUT side of the D relay, so the digital drive and sense circuits are not exposed to high voltages.

U25, an 8-bit latch, can be set by software command to engage any combination of 8 relays. The programmer must make sure that these relays are disengaged when not being used. The voltages that appear on the DIRECT lines may be outside the range of voltages tolerable by the driver circuits on the DR2p board.

Every third wire in the cable is grounded. Every signal wire is therefore adjacent to a ground, and thereby protected from inductive crosstalk.

## Gate Array Output Multiplex

The output multiplex function allows DFP to control channels directly. Except the eight channels which are DIRECT functions, the gate array outputs of 24 of the 32 channels are multiplexed in opposition to DFP functions. Of the 24 multiplexed channels, not all are done in the same way.

Channels 0–7 (“Port A” channels) and channels 8–15 (“Port B” channels) are bytewise bidirectional, and can be enabled by signals from the DUT. Channels 16–23 (“Port C” channels) are bitwise bidirectional, and have additional functions, such as the strobes that enable the Port A and Port B channels, that are unlike normal microprocessor I/O ports.

### Port A and B Channel Groups

When the DR2p is in the NORMal mode (ENORM* low), buffer U5 pin 9 drives the pullup PNP transistor Q6, and buffer U2 pin 9 drives the pulldown NPN transistor Q22. This provides a signal path from the gate array outputs to the amplifier in normal in-circuit testing.

When ENORM* is high, these buffers are turned off, and the gate-array path is disabled. Signals DH07* and DL07 will take over control. The buffers in this case are U4 pin 9 and U3 pin 9. Assuming for the moment that EPT* is low, these two buffers, instead of the buffers on the gate array outputs, will drive the two amplifier transistors.

These buffers’ inputs are supplied, respectively, by NAND gate U28 pin 3 and AND gate U40 pin 3. These two gates are active only if the signal EPT 0–7 is high. Thus data stored in latch U53 will be available on the DR2p, but not driven onto the tester channels until EPT 0–7 is true.

DFP can set EPT 0–7 to a constant true state (for example, in the ADDRESS mode, where addresses are to be constantly enabled) or can arrange to have EPT 0–7 controlled by a strobe signal coming from the DUT. In the latter case, the strobe signal is brought in on one of the Port C channels.

A full discussion of EPT 0–7 is found in the section titled “U74—the I/O Control PAL” later in this chapter. The important message in the current section is that the Port A channels and the Port B channels operate bytewise. It is impossible to make one bit an input and simultaneously make one of its neighbors an output. It is possible, however, to make Port A an input and port B an output, or vice versa.

Inputting from Ports A and B is done using U55 and U52. The readback operation can occur while the channel amplifiers are driving. This feature is used in self testing, but may lead to difficult troubleshooting of programs in development. If your program is to read back data from the DUT on Port A or Port B, be sure to disable the driver group in question or you will simply read back what you are outputting.

## Port C Channel Group

Whereas Ports A and B operate bytewise, Port C operates bitwise. The Port C channel group also has functions that, depending on the mode control, go beyond simple inputting and outputting of bits to and from the DFP microprocessor. The serial port lines and the handshake lines flow through the Port C channel group as well.

Each bit of the enable latch serves to enable its associated driver amplifier. Thus it is possible to enable or disable the driver amplifiers of channels 16 through 23 a bit at a time.

Each bit of the data latch U58, with the exception of channels 16 and 20, serves to determine the high-or-low state of its driver if that driver is enabled. Thus, with the same two exceptions, DFP can set any bit of Port C high, low, or high impedance.

Channels 16 and 20 are used either for serial data or for handshake purposes. Drivers 16 and 20 are enabled by the corresponding bits of the enable latch, but the high-or-low drive state is determined by signals PT16R (for channel 16) and PT20R (for channel 20). These signals come from the Address Decode PAL U73, and will be discussed below in the section on U73.

Note that the bits of the data latch that correspond to channels 16 and 20 are inputs to that same PAL. This suggests a mode in which channels 16 and 20 behave like the other six channels in the Port C group.

Readback of the Port C channel group is accomplished by a tandem pair of mux gates U45 and U56. The programmer is reminded that this operation is capable of reading back the states of the channel driver amplifiers if they happen to be enabled. That will override any signal the DUT is trying to generate. Be sure to pay attention to the Port C enable latch.

The four signals of special interest originating on this page are I17, I18, I21 and I22—the detected signals coming from the DUT on channels 17, 18, 21 and 22 respectively. They are used in handshaking and in the receipt of serial data from a DUT serial port. Handshaking and serial data bit timings are too precise to allow the DFP computer to handle them by sampling in software—they take a more direct route.

I21 and I17 pass through U59 and go directly to J4 (the 80-pin ribbon cable connector) pins 63 and 62. A signal coming from the CCC via J4 pin 48 controls selection of this path. This path is used when the CCC needs to receive serial asynchronous data from the DUT. The alternative is for the CCC to receive EPT 8–15 and EPT 0–7, which are part of the handshake logic. In both instances, the CCC has a path through which it can receive these real-time signals directly, without any time distortion due to latching or sampling latency.

## U73—the Address Decode PAL

Description of the functions of U73 follows the listing of the Boolean equations below. The equations are expressed in PALASM language.

```
; Programmed part is Teradyne P/N 46248 (046-248)

CHIP ADDR_DECODE PAL16L8               ; Production unit in
GAL16V8

; 1    2    3     4    5     6    7     8     9      10

  A0   A1   1E   PT20  PT16  ARDY BRDY  DIR  GBUSIN  GND


; 11      12     13    14    15    16    17     18     19    20

  NC  /READBACK /ES4  /ES3  /ES2  /ES1  /ES0   PT20R  PT16R
VCC


;               Address | Register

;               --------|--------------------------

;               0          | PortA data

;               1          | PortB data

;               2          | PortC data

;               3          | PortC enable

;               4          | I/O Control reg

;               5          | PortA and PortB data

;               6          | Address

;               --------|--------------------------


EQUATIONS


PT16R   = PT16

        + ARDY


PT20R   = PT20

        + BRDY


/READBACK       = DIR

               + GBUSIN
```

```
ES0     = /A0 * /A1 * /1E              ; PORT A DATA   0
       +  A0 * /A1 *  1E             ;                   5
       + /A0 *  A1 *  1E             ;                   6

ES1     =  A0 * /A1 * /1E              ; PORT B DATA   1
       +  A0 * /A1 *  1E             ;                   5
       + /A0 *  A1 *  1E             ;                   6

ES2     = /A0 *  A1 * /1E              ; PORT C DATA   2
       + /A0 *  A1 *  1E             ;                   6

ES3     =  A0 *  A1 * /1E              ; PORT C ENABLE  3

ES4     = /A0 * /A1 *  1E              ; I/O CONTROL REG  4
```

PT16R is either the first bit of the Port C data register (it is signal PT16 because it relates to GBUS<16>) or the level ARDY. ARDY is used in handshaking and in serial data transmission to the DUT. Be sure to set PT16 low when using handshaking or when sending serial data. If PT16 remains high, the ARDY (TXDA) signal is masked, and haphazard data transfer to the DUT CPU will result. PT20R works the same way as PT16R, except with respect to GBUS<20> and the BRDY (TXDB) signal.

The /READBACK signal, called READBACK* in the schematic, is asserted low when DIR and GBUSIN both are low. GBUSIN, controlled by logic on the CCC, is high when the CCC is transmitting to the GBUS and low when taking data from the GBUS. READBACK places 24 bits of data from the DUT on the GBUS via the bus input multiplexers U56 , U55 and U52. READBACK is also used in selftest, to allow a CCC to read back data from the Port A, B, and C channel amplifiers. In most applications, be sure to disable the channel drivers on the DUT lines that you wish to read back.

ES0 through ES4 decode addresses of data latches on DR2p. Some latches have multiple addresses, e.g., Port A data can be affected by writes to address 0, 5 or 6. Address 0 writes to Port A alone. Addresses 5 and 6 write to both Port A and Port B latches. These equations implement the logic of the register maps given earlier in this document. The PTLATCH signal, which is common to all latches, actuates the latches on DR2p. The latches themselves respond or not according to their individual addresses as decoded by U73. Below is a list of latches.

| | | |
|---|---|---|
| U53: | ES0: | Port A data |
| U54: | ES1: | Port B data |
| U58: | ES2: | Port C data |
| U57: | ES3: | Port C enable |
| U76 and U26: | ES4: | mode controls, DIRECT relays |

## U74—the I/O Control PAL

Description of the functions of U74 follows the listing of the Boolean equations below. The equations are expressed in PALASM language.

```
; Programmed chip is Teradyne p/n 46247 or 046-247

CHIP IO_CONTROL PAL16L8         ; Production unit in GAL16V8


;  1    2    3    4    5    6    7    8    9    10

  PAO0 PAO1 PAI0 PAI1 PBO0 PBO1 PBI0 PBI1 I22  GND


;  11   12      13     14   15   16   17    18     19      20

  I21  LATCHB EPT8-15 I18  I17  NC1  NC2  LATCHA  EPT0-7  VCC


EQUATIONS


EPT0-7  =  /PAO0 * /PAO1               ; PORT A STATIC OUTPUT

        +  /PAO0 *  PAO1 * /I17         ; PORT A DYNAMIC, OE
LOW

        +   PAO0 * /PAO1 *  I17         ; PORT A DYNAMIC, OE
HIGH


EPT8-15 =  /PBO0 * /PBO1               ; PORT B STATIC OUTPUT

        +  /PBO0 *  PBO1 * /I21         ; PORT B DYNAMIC, OE
LOW

        +   PBO0 * /PBO1 *  I21         ; PORT B DYNAMIC, OE
HIGH


LATCHA  =  /PAI0 * /PAI1                ; PORT A TRANSPARENT
INPUT

        +  /PAI0 *  PAI1 * /I18          ; PORT A STROBE LOW

        +   PAI0 * /PAI1 *  I18          ; PORT A STROBE HIGH


LATCHB  =  /PBI0 * /PBI1               ; PORT B TRANSPARENT
INPUT

        +  /PBI0 *  PBI1 * /I22          ; PORT B STROBE LOW

        +   PBI0 * /PBI1 *  I22          ; PORT B STROBE HIGH
```

Pins 1 through 8 are latched in U76 and do not have explicit names in the schematic. However, these bits are named in the discussion of Block 1, Offset 2 earlier in this
document.

The information latched in U76 implements the input and output modes of Ports A and B. This latch is combined in a 16-bit word with U26, which controls the relays. A 16-bit write operation is required with the relay information in the high byte. An attempt at an 8-bit write to U76 will disturb the relay settings and be very difficult to debug.

Port A and Port B have three output modes and three input modes each.

**Output Modes**

The output portion of the port can be “static,” i.e., its EPT signal constant, causing the driver amplifiers to be on at all times. The output portion of the port can also be “dynamic,” i.e., its EPT signal can be controlled by a signal originating on the DUT, allowing the DUT CPU to view the port as a virtual input port on its own data bus. The enabling signal for output Port A is I17. I17 comes from the DUT on channel 17. It is resolved as RESP<17> at U45.

The CCC can read I17 conventionally as part of the Port C channel group. The logic in this PAL also lets I17 be used as a strobe to enable the Port A drivers to implement the virtual DUT input port function. According to the bit pattern written into latch U76, I17 can be a high-going enable (i.e., enabling the Port A amplifiers when I17 is high) or a low-going enable.

Just as signal I17 serves Port A as an enable and is controlled by bits PAO0 and PAO1, signal I21 serves Port B and is controlled by bits PBO0 and PBO1. Port A and Port B do not have to be in the same output mode, i.e., one can be dynamic while the other is static, or one can be dynamic with a high strobe while the other is dynamic with a low strobe, and so on.

**Input Modes**

The input sides of these two ports also have three modes. First, the port can be “transparent.” Any time the CCC reads the port, the current conditions of the port’s eight channels will be transferred to the DFP computer. The LATCHA signal (LATCHB, if discussing Port B) is high as it goes to the input muxes U52 (U55 for LATCHB). The READBACK* signal allows these muxes to drive the GBUS.

DFP also allows either Port A or B, or both, to serve as virtual DUT output ports. In this case, the port’s state can be latched by a signal coming in from the DUT. In this mode, the LATCHA or LATCHB signal is low most of the time. The signal is pulsed high only when the state is to be latched. DFP can then use the READBACK* signal to read the latched data.

In the latched modes, LATCHA follows the DUT-originated signal I18 or its complement, depending on the states of PAI0 and PAI1. Similarly, LATCHB follows I22 or its complement depending on the states of PBI0 and PBI1. I18 and I22 come from the DUT on channels 18 and 22 respectively, whose signals are resolved to RESP18 and RESP22. These channels are associated with corresponding channel drive amplifiers in the Port C group. Be sure to disable the channel driver amplifier on any channel you wish to use to listen to a DUT signal.

## Significant Features of Handshaking and Serial Ports

The bulk of the circuitry associated with handshaking and serial interchanges is located on the CCC, as discussed in the Channel Control Card section of this chapter. Below is a summary of the significant features of the DR2p.

- The DR2p channels relevant to handshaking and serial data exchange are in the Port C group, i.e., 16 to 23.
- The two relevant high-speed, nonlatched signals that go from the CCC to DR2p are J4-45 and J4-47. They are labeled ARDY and BRDY. They carry handshake information (i.e., “Port A is Ready”) in the handshake mode, and carry a serial asynchronous bitstream in serial mode.
- The two relevant high-speed, nonlatched signals that go from the DR2p to the CCC are J4-63 and J4-62. The CCC, using a signal on J4-48, chooses whether to connect them to I21 and I17 (serial bitstreams coming from the DUT) or to EPT8-15 and EPT0-7 (handshake signals noting the fact that the DUT has enabled the Port A or Port B channel driver amplifier group).
- These two pairs of high-speed signals are used either in handshaking or in serial data exchange, but not both simultaneously. If the EPT lines are selected at U59, handshaking is in use. If I17 and I21 are selected at U59, serial data exchange is in use.
- Port C, like Port A and Port B, can drive data out to the DUT as well as receive it. Be sure to disable the Port C enable bits for the channels you use as inputs from the DUT.

## Logic Design—Channel Control Card

The Channel Control Card is an ISA-bus board that occupies part of the input-output space of the DFP computer. It contains registers, counters, serial peripheral ports, timers and miscellaneous logic that controls the DR2p board. There is one CCC for each DR2p. They are connected to each other by an 80-conductor ribbon cable. DFP as a system can support up to four such pairs.

Different CCC/DR2p pairs in a system can be in different modes. In a flash-in-free-air application, for example, one pair is set up as an address counter and another pair is set up to exchange data in parallel bytes.

Most software operation in the DFP computer is polled rather than interrupt-driven.

### Connectors, Signal Names, and Cable Pins

The edge fingers which plug into the computer motherboard are labeled A, B, C and D. The 80-pin connector labeled J1 connects pin for pin with the 80-pin connector labeled J4 on the DR2p. Some signal names on the DR2p are not exact matches for the names of the corresponding functions on the CCC. Most cable pin signals are immediately buffered on the DR2p and are not named until buffered. In the following table, the buffered names are used where appropriate names do not exist for the raw signals.

**Table 3.5 Cable Pin Numbers, Signal Names**

| Cable Pin Number | DR2p Name (buffered as appropriate) | CCC name |
|---|---|---|
| 1 | gnd1 | gnd |
| 2 | GBUS<0> | P0 |
| 3 | GBUS<1> | P1 |
| 4 | gnd | gnd |
| 5 | GBUS<2> | P2 |
| 6 | GBUS<3> | P3 |
| 7 | gnd | gnd |
| 8 | GBUS<4> | P4 |
| 9 | GBUS<5> | P5 |
| 10 | gnd | gnd |
| 11 | GBUS<6> | P6 |

**Table 3.5 Cable Pin Numbers, Signal Names**

| Cable Pin Number | DR2p Name (buffered as appropriate) | CCC name |
|---|---|---|
| 12 | GBUS<7> | P7 |
| 13 | gnd | gnd |
| 14 | GBUS<8> | P8 |
| 15 | GBUS<9> | P9 |
| 16 | gnd | gnd |
| 17 | GBUS<10> | P10 |
| 18 | GBUS<11> | P11 |
| 19 | gnd | gnd |
| 20 | GBUS<12> | P12 |
| 21 | GBUS<13> | P13 |
| 22 | gnd | gnd |
| 23 | GBUS<14> | P14 |
| 24 | GBUS<15> | P15 |
| 25 | gnd | gnd |
| 26 | GBUS<16> | P16 |
| 27 | GBUS<17> | P17 |
| 28 | gnd | gnd |
| 29 | GBUS<18> | P18 |
| 30 | GBUS<19> | P19 |
| 31 | gnd | gnd |
| 32 | GBUS<20> | P20 |
| 33 | GBUS<21> | P21 |
| 34 | gnd | gnd |
| 35 | GBUS<22> | P22 |
| 36 | GBUS<23> | P23 |
| 37 | gnd | gnd |
| 38 | GBUSENABLE* | BEN* |
| 39 | DIR | BIE* |
| 40 | gnd | gnd |
| 41 | PTLATCH | PCK |
| 42 | gnd | gnd |
| 43 | gnd | gnd |
| 44 | GBUSIN | BDIR |
| 45 | ARDY | ARDY |
| 46 | gnd | gnd |
| 47 | BRDLY | BRDY |
| 48 | no name | CSEL |

**Table 3.5 Cable Pin Numbers, Signal Names**

| Cable Pin Number | DR2p Name (buffered as appropriate) | CCC name |
|---|---|---|
| 49 | gnd | gnd |
| 50 | A0 | PA0 |
| 51 | A1 | PA1 |
| 52 | gnd | gnd |
| 53 | 1E | PA2 |
| 54 | ESY* | PESY* |
| 55 | gnd | gnd |
| 56 | DSY* | PDSY* |
| 57 | ISTB* | PIST* |
| 58 | gnd | gnd |
| 59 | DSTB* | PDST* |
| 60 | EPT* | PPT* |
| 61 | gnd | gnd |
| 62 | EPT0-7 | PCA |
| 63 | EPT8-15 | PCB |
| 64 | gnd | gnd |
| 65 | Vpp | Vpp |
| 66 | Vpp | Vpp |
| 67 | +18 | +18 volts |
| 68 | +18 | +18 volts |
| 69 | -18 | -18 volts |
| 70 | DIRECT25 | TX232A |
| 71 | DIRECT26 | RX232A |
| 72 | gnd | gnd |
| 73 | DIRECT27 | TX232B |
| 74 | DIRECT28 | TX232B |
| 75 | gnd | gnd |
| 76 | DIRECT29 | n/c DX <29> |
| 77 | DIRECT30 | n/c DX<30> |
| 78 | gnd | gnd |
| 79 | DIRECT31 | n/c |
| 80 | not used | SP1 |

## CCC Address Management

Each CCC uses several addresses, as described below. These must not overlap or collide with any other addresses used in the DFP computer, or unpredictable operation will result.

The DFP computer will see each CCC at a set of addresses governed by the DIP Switch SW1. An open switch represents a “true” bit. Note that only bits 3 through 9 of the address bus participate in this comparison.

Address bits 1, 2, 12, 13, and 14 are used on the CCC as well. Bits 0, 10 and 11 are not used at all.

Bits 1 through 2 are referred to as the “Offset.”
Bits 3 through 9 are referred to as the “Card address.”
Bits 12 through 14 are referred to as the “Block address.”

The address of an individual register has contributions from the Card address, the Block address, and the Offset.

The following diagram illustrates the assignment of address bits to functionalities. B represents a Block, C a Card, O an offset, and X a don’t care.

```
msb ----------> lsb
xBBB xxCC CCCC COOx
```

### Offset Contribution

The Offset address has two bits, allowing four possible values. Because the LSB of the physical address is not used, the Offset contribution can have the following values:

Block and Card are both zero.

0x0000
0x0002
0x0004
0x0006

For example, to place the CCC 0 (zero) at address 0x5384, set the board address switches as follows: choose Block address 5 and Offset address 4. The addition of all these contributions looks like Table 3.6 below. A dot “.” represents “don’t care.”

**Table 3.6 Example of Board Address Settings**

| ADDRESS bit position | 15 | 14 | 13 | 12 | 1 1 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SW1 Pin number | | | | | | | 8 | 7 | 6 | 5 | 4 | 3 | 2 | . | | | |
| | . | . | . | . | . | . | 1 | 1 | 1 | 0 | 0 | 0 | 0 | . | . | . | = 0x0380 |
| BLOCK Address 5 | . | 1 | 0 | 1 | . | . | . | . | . | . | . | . | . | . | . | . | = 0x5000 |
| OFFSET Address 4 | . | . | . | . | . | . | . | . | . | . | . | . | . | 1 | 0 | . | = 0x0004 |
| Sum of all three contributions | | | | | | | | | | | | | | | | | =0x5384 |

To troubleshoot the recognition of a Card address, probe pin 19 of U3. You need an extender, as this chip is next to the A connector.

## Card Contribution

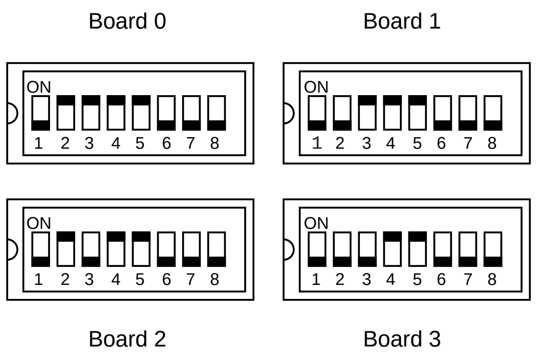
The Card address has seven bits, allowing 128 possible board addresses. With Block and Offset addresses both zero, the Card address contribution can have the following range:

| | |
|---|---|
| Lowest possible Card address: | 0x0000 |
| Highest possible Card address: | 0x03F8 |

The pre-defined Card addresses are as follows:

```
0x0380 = card 0    0x0390 = card 2
0x0388 = card 1    0x0398 = card 3
```

Figure 3.5 DIP Switch Settings



**Note: SW1, position 1 must always be off.**

## Block Contribution

The Block address has three bits, allowing eight values. Although they are referred to as Blocks 0 through 7, these numbers actually represent contributions to the physical addresses as follows:

Card and Offset are both zero.

|   |   |
|---|---|
| 0 | 0x0000 |
| 1 | 0x1000 |
| 2 | 0x2000 |
| 3 | 0x3000 |
| 4 | 0x4000 |
| 5 | 0x5000 |
| 6 | 0x6000 |
| 7 | 0x7000 |

## Channel Control Card Hardware Details

The following sections discuss Channel Control Card hardware.

### The IBUS and Its Controls

An internal 16-bit bus called IBUS buffers the computer motherboard's data bus. Chips U1 and U2 are bidirectional buffers which are enabled by the recognition of the Card address. Their direction is governed by the RD* signal, which is a buffered version of the motherboard signal -IOR entering the CCC at pin B 14. Data flows to the motherboard when RD* is low, and from the motherboard when RD* is high.

An array of PAL chips U6 through U10 governs the selection of the various registers into which data is written on write cycles and from which data is read on read cycles.

A delay system composed of U12A, U13, and U11 produces a delayed write strobe to the DR2p when DR2p registers are being written.

### Counters Used as Output Port Registers

Counters U14 through U19 are used as latches for Ports A, B, and C, simplifying and accelerating the generation of addresses in flash-in-free-air applications. The computer can produce the next address simply and quickly using a small number of instructions.

Refer to Table 3.3 (Programmer's IO Port Map) earlier in this chapter, and note that Block 0 operations affect these counters. The text following the Port Map explains how to control them.

The hardware realization involves a small number of signals: WR* coming from U4 pin 7 clocks the counters. PD1 is a low signal from the pulldown resistor. If E3* from U9 pin 16 is low, the counters will count. The UpDown (UD) signal, which originates at U39 pin 6, determines the direction.

## Port A and Port B

Ports A and B operate as described in the section titled “Register Descriptions: Block 0 - Address Counter” earlier in this chapter. To load all 24 bits of the starting address, first write the 16 bits corresponding to Ports A and B at offset 0, then write the MS byte to Port C at offset 2. To load only 16 bits of starting address into Ports A and B, write instead to offset 4.

## Port C

The Port C channel group is the upper 8 bits of the 24 bit counter. Its behavior is different from that of Ports A and B. Port C is bit-selected and contains handshake and serial capabilities.

## Serial Port, Logic Levels, and RS232 Levels

The Zilog/Hitachi 85C30 is equipped with a 7.3728 MHz crystal. The 9.216 MHz crystal shown on some schematics is not installed. The location is left open for you to customize if necessary. If you have not worked with the 85C30 or its relatives before, Teradyne advises you to contact Hitachi or Zilog and obtain the programmer’s guides for this sophisticated and often puzzling chip. You can obtain some understanding of it from studying Teradyne example programs, but those programs use only a small fraction of this chip’s capabilities.

DFP is equipped to talk to DUT serial asynchronous ports using either “logic” levels (i.e,. 5 volts mark, 0 volts space) or EIA RS232 levels ( +15>mark>+5, -5>space>-15). Each CCC/DR2p pair has two transmitters and two receivers. Transmissions at logic level flow to the DUT through the channel drive amplifiers. Reception at logic levels flows to the 85C30 through mux gates and PAL chips. Transmissions at RS232 level flow to the DUT through relay-connected DIRECT channels on the DR2p. Channel driver amplifiers are not used, because they cannot accommodate the necessary voltage swings. Translation to RS232 levels is done with U32, a MAX242 level shifter chip. The MAX generates its own high and low voltages.

Reception at RS232 levels also flows through the DIRECT channels and uses the MAX242 chip to convert back to logic levels.

The logic level transmissions take the following route: the 85C30’s transmitter outputs (pins 16 and 29 of U36) drive inputs to the PAL U33, as well as the inputs to the level shifter U32. TDA and TDB pass through the jumper block J2 to become TDA1 and TDB1. They are sent to DR2p as ARDY and BRDY on J1 pins 45 and 47.

Received logic level signals enter the CCC on J1 pins 62 and 63, PCA and PCB. Chip U26 converts these to AACK1 and BACK1. These signal lines are used alternatively in handshaking, and one CCC cannot do both handshaking and serial communication at the same time. After passing through jumper block J2, the signals are called AACK and BACK. They enter the PAL U33. See equations above for U33 output pins 12 and 13, RXDA and RXDB. The PAL selects whether the 85C30 will obtain its RXD signals from the MAX242 or from the AACK and BACK lines.

## Handshake Flipflops

The two signaling flipflops U30A and U30B work identically, so only the A side is described here. When handshaking is established, the flipflop is set every time the CCC writes to the Port A data register on the DR2p. The signal RDYA goes high, indicating to the DUT that fresh data is available to take.

Immediately after writing this byte in the Port A data latch, the DFP CPU can look up the next, thereby eliminating overhead. It must not write it, however, until it finds that DUT CPU has cleared the signaling flipflop.

RDYA's path to the DUT is as follows: RDYA passes through PAL U33 and emerges as TDA. (Remember, these lines can be used for serial data when they are not being used for handshaking.) TDA passes through jumper block J2 and becomes TDA1. TDA1 is buffered by U29, lower center. The buffered signal is known as ARDY, and is sent to DR2p on J1 pin 45.

The discussion of the DR2p, above, traced this signal out to Port C channel 16 via the signal PT16R. The DUT CPU, by running a program designed for the purpose, knows that fresh data is available.

The DUT CPU inputs the data onto its data bus by generating a pulse of the appropriate polarity on the DR2p's channel 17. The DUT CPU, having taken the data, can write the data into flash memory, a process that takes several microseconds. The pulse the DUT used to input the data becomes the signal EPT0-7 as was discussed in the section on the DR2p board earlier in this chapter. EPT0-7 finds its way to pin 62 of the 80-pin cable. It arrives at the CCC where it gets the new name "PCA."

PCA is buffered by U26 and gets another new name "AACK1." AACK1 passes through jumper block J2 and changes its name, for the last time, to AACK. AACK goes to the gate U5A and resets the signaling flipflop. Thus, the signaling flipflop is reset when the DUT CPU inputs the data that was placed in the Port A data latch.

By this time, the DFP CPU has looked up the next byte for Port A and is ready to write it. Before writing, it must check to see that the flipflop has been cleared. In addition to signaling the DUT CPU through the path described, the flipflop signals the DFP CPU as well. The RDYA signal enters PAL U33 and emerges as TDA, becoming TDA1. The DFP CPU can read the state of TDA1 by means of input port U28. A DFP program can loop on this bit and write the new data as soon as the old data has been taken. That way, neither processor waits very long for the other. Such a loop should include a timeout feature to prevent hangups in the case of DUT failures.

## Jumper Block and Custom Daughterboard Provision

The jumper block J2, referred to in the descriptions above, is a 34-pin header that normally has jumpers on it to connect certain signal lines from one side to the other. Its purpose is to allow small amounts of customization to handle special cases without creating entirely new versions of the CCC. You can design a small custom daughterboard to mate with this connector, or you can use a 34-conductor ribbon cable to connect the CCC with a larger custom daughterboard located further from the CCC.

Normal jumper configuration is shown in the table following.

**Table 3.7 Normal Jumper Configuration**

| Signal | Pin # | Connection | Pin # | Signal |
|---|---|---|---|---|
| TX232A -> J1-70 | 1 | ----jpr---- | 2 | T1OUT |
| RX232A -> J1-71 | 3 | ----jpr---- | 4 | R1IN |
| gnd | 5 | PC trace | 6 | gnd |
| TX232B -> J1-73 | 7 |  | 8 | T2OUT |
| RX232B -> J1-74 | 9 |  | 10 | R2IN |
| gnd | 11 | PC trace | 12 | gnd |
| DX<29> -> J1-76 | 13 |  | 14 | n/c |
| DX<30> -> J1-77 | 15 |  | 16 | n/c |
| TDA | 17 | ----jpr---- | 18 | TDA1 |
| TDB | 19 | ----jpr---- | 20 | TDB1 |
| gnd | 21 | PC trace | 22 | gnd |
| AA CK | 23 | ----jpr---- | 24 | AACK1 |
| BACK | 25 | ----jpr---- | 26 | BACK1 |

| Signal | Pin # | Connection | Pin # | Signal |
|---|---|---|---|---|
| gnd | 27 | PC trace | 28 | gnd |
| +18 volts | 29 | PC trace | 30 | +18 volts |
| -18 volts | 31 | PC trace | 32 | -18 volts |
| +5 volts | 33 | PC trace | 34 | +5 volts |

## DAC and Vpp

Many nonvolatile technologies require an elevated voltage at one pin in order to write. Some CPU families, e.g., the Motorola 68HC05 family, require a single 9-volt signal to start their bootstrapping operations. DFP provides a programmable voltage source for this purpose. Please note that the ±18 volt supplies which power this part of the circuitry originate in the tester and find their way to the CCC from the DR2p over the 80-conductor ribbon cable. They do not originate in the DFP computer.

## Timers

A timer chip 82C53 is provided to simplify timeout programming. See Intel's programming guides for information about this IC.

## PAL Equations

Below are the PAL equations for the Channel Control Cards.

### U7 Pattern 0320

```
Title    PT2 Address Decode PAL
Pattern  0320-3
Revision 3.0

CHIP ADDR4 PAL16L8
; Production unit in GAL16V8
; Provides misc address decodes.
; Name    Pin   Dir     Function
; -------------------------------------------------------------------------
  A2    ; 1     IN      Address 2
  A1    ; 2     IN      Address 1
  A12   ; 3     IN      Address 12
  A13   ; 4     IN      Address 13
  A14   ; 5     IN      Address 14
 /IOW   ; 6     IN      IOW
 /IOR   ; 7     IN      IOR
  NC1   ; 8     IN      No function.
  PCK   ; 9     IN      DR2 ck signal.
  GND   ; 10    GND     GND
 /GRP   ; 11    IN      Card enable signal.
 /WS    ; 12    OUT     Wait state indicator - Default-wait/No-wait.
 /WEN   ; 13    OUT     Enable OWS/IO_CH_RDY/IO_CS_16 to bus.
 /TRG2  ; 14    OUT     Trigger CTC channel 2.
 /TRG1  ; 15    OUT     Trigger CTC channel 1.
 /TRG0  ; 16    OUT     Trigger CTC channel 0.
  NWS   ; 17    OUT     No wait-states signal to bus.
  NC3   ; 18    OUT     No function.
  NC4   ; 19    OUT     No function.
  VCC   ; 20    VCC     VCC
; -------------------------------------------------------------------------

EQUATIONS

  WS    = /A12 * /A13 *  A14 *              GRP         ; 4.X wait if CTC
        +  A12 * /A13 *  A14 *              GRP         ; 5.X wait if SIO
       +  A12 *  A13 * /A14 *            GRP * IOR   ; 3.X wait if READ PORTS
 /NWS   = /A12 * /A13 *  A14 *              GRP         ; 4.X wait if CTC
        +  A12 * /A13 *  A14 *              GRP         ; 5.X wait if SIO
       +  A12 *  A13 * /A14 *            GRP * IOR   ; 3.X wait if READ PORTS
  WEN   =                                   GRP * IOR   ; read
        +                                   GRP * IOW   ; write

TRG0  =  A12 *  A13 *  A14 * /A2 * /A1 *  GRP * IOW   ; W 7.0
TRG1  =  A12 *  A13 *  A14 * /A2 *  A1 *  GRP * IOW   ; W 7.2
TRG2  =  A12 *  A13 *  A14 *  A2 * /A1 *  GRP * IOW   ; W 7.4
```

## U8 Pattern 0321

```
Title    PT2 Address Decode PAL
Pattern  0321-1
Revision 1.0

CHIP ADDR3 PAL16L8              ; Production unit in GAL16V8
; Provides misc address decodes etc. for DR2 card.
; Name    Pin   Dir     Function
; ------------------------------------------------------------------------
  A2    ; 1     IN      Address 2
  A1    ; 2     IN      Address 1
  A12   ; 3     IN      Address 12
  A13   ; 4     IN      Address 13
  A14   ; 5     IN      Address 14
 /IOW   ; 6     IN      IOW
 /IOR   ; 7     IN      IOR
  NC1   ; 8     IN      No function.
  PCK   ; 9     IN      DR2 ck signal.
  GND   ; 10    GND     GND
 /GRP   ; 11    IN      Card enable signal.
  SP1   ; 12    OUT     No function.
  PA2   ; 13    OUT     Address PA2 to DR2.
  PA1   ; 14    OUT     Address PA1 to DR2.
  PA0   ; 15    OUT     Address PA0 to DR2.
 /PDST  ; 16    OUT     DR2 gate array data strobe.
 /PIST  ; 17    OUT     DR2 gate array instruction strobe.
 /PDSY  ; 18    OUT     DR2 gate array data sync.
 /PESY  ; 19    OUT     DR2 gate array enable sync.
  VCC   ; 20    VCC     VCC
; ------------------------------------------------------------------------

EQUATIONS

PA0   =  A12 * /A13 * /A14 * /A2 * /A1 *  GRP         ; 1.0
        +  A12 *  A13 * /A14 * /A2 *  A1 *  GRP         ; 3.2
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP         ; 3.6

PA1   = /A12 * /A13 * /A14 * /A2 *  A1 *  GRP         ; 0.2
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP         ; 0.4
        + /A12 * /A13 * /A14 *  A2 *  A1 *  GRP         ; 0.6
        +  A12 * /A13 * /A14 * /A2 * /A1 *  GRP         ; 1.0
        +  A12 *  A13 * /A14 *  A2 * /A1 *  GRP         ; 3.4

PA2   = /A12 * /A13 * /A14 * /A2 *  A1 *  GRP         ; 0.2
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP         ; 0.4
        + /A12 * /A13 * /A14 *  A2 *  A1 *  GRP         ; 0.6
        +  A12 * /A13 * /A14 * /A2 *  A1 *  GRP         ; 1.2
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP         ; 3.6

PIST  = /A12 *  A13 * /A14 * /A2 * /A1 *  GRP * /PCK  ; W 2.0

PESY  = /A12 *  A13 * /A14 * /A2 *  A1 *  GRP * /PCK  ; W 2.2

PDSY  = /A12 *  A13 * /A14 *  A2 * /A1 *  GRP * /PCK  ; W 2.4

PDST  = /A12 *  A13 * /A14 *  A2 *  A1 *  GRP * /PCK  ; W 2.6
```

## U9 Pattern 0322

```
Title    PT2 Address Decode PAL
Pattern  0322-2
Revision 2.0

CHIP ADDR2 PAL16L8              ; Production unit in GAL16V8
; Provides misc address decodes.
; Name    Pin   Dir     Function
; -----------------------------------------------------------------------
A2    ; 1     IN      Address 2
A1    ; 2     IN      Address 1
A12   ; 3     IN      Address 12
A13   ; 4     IN      Address 13
A14   ; 5     IN      Address 14
/IOW   ; 6     IN      IOW
/IOR   ; 7     IN      IOR
NC1   ; 8     IN      No function.
PCK   ; 9     IN      DR2 ck signal.
GND   ; 10    GND     GND
/GRP   ; 11    IN      Card enable signal.
/E7    ; 12    OUT     Misc register ck.
/E6    ; 13    OUT     Port C I/P data enable.
/E5    ; 14    OUT     Port B I/P data enable.
/E4    ; 15    OUT     Port A I/P data enable.
/E3    ; 16    OUT     Address counter enable.
/E2    ; 17    OUT     Counter MSB/Port C write enable.
/E1    ; 18    OUT     Counter MID/Port B write enable.
/E0    ; 19    OUT     Counter LSB/Port A write enable.
VCC   ; 20    VCC     VCC
; -----------------------------------------------------------------------

EQUATIONS

  E0    = /A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 0.0
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOW   ; W 0.4
        +  A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 1.0
        +  A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 1.2
        + /A12 *  A13 * /A14 *              GRP * IOW   ; W 2.X
        +  A12 *  A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 3.0
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 3.6

  E1    = /A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 0.0
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOW   ; W 0.4
        +  A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 1.2
        + /A12 *  A13 * /A14 *              GRP * IOW   ; W 2.X
        +  A12 *  A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 3.2
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 3.6

  E2    = /A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 0.2
        +  A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 1.0
        +  A12 *  A13 * /A14 *  A2 * /A1 *  GRP * IOW   ; W 3.4

  E3    = /A12 * /A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 0.6
  E4    =  A12 *  A13 * /A14 * /A2 * /A1 *  GRP * IOR   ; R 3.0
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOR   ; R 3.6
        + /A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOR   ; R 0.0
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOR   ; R 0.4

  E5    =  A12 *  A13 * /A14 * /A2 *  A1 *  GRP * IOR   ; R 3.2
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOR   ; R 3.6
        + /A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOR   ; R 0.0
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOR   ; R 0.4

  E6    =  A12 *  A13 * /A14 *  A2 * /A1 *  GRP * IOR   ; R 3.4
        + /A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOR   ; R 0.2

  E7    = /A12 *  A13 *  A14 *  A2 *  A1 *  GRP * IOW   ; R 6.6
```

## U10 Pattern 0323

```
Title    PT2 Address Decode PAL
Pattern  0323-4
Revision 4.0

;Revised 2/18/94
;Changed the sense of W16 from /W16 to W16
;Made reads from port C (3.4) 16 bit.
;Dropped SIO (5.X) from W16.

CHIP ADDR1 PAL16L8              ; Production unit in GAL16V8

; Provides misc address decodes.

; Name    Pin   Dir     Function
; --------------------------------------------------------------------------
  A2    ; 1     IN      Address 2
  A1    ; 2     IN      Address 1
  A12   ; 3     IN      Address 12
  A13   ; 4     IN      Address 13
  A14   ; 5     IN      Address 14
 /IOW   ; 6     IN      IOW
 /IOR   ; 7     IN      IOR
  NC1   ; 8     IN      No function.
  PCK   ; 9     IN      DR2 ck signal.
  GND   ; 10    GND     GND
 /GRP   ; 11    IN      Card enable signal.
 /STAT  ; 12    OUT     Status read enable.
  W16   ; 13    OUT     16 bit transfer acknowledge signal.
 /DAC   ; 14    OUT     DAC write ck.
 /SIO   ; 15    OUT     SIO write ck.
 /CTC   ; 16    OUT     CTC write ck.
 /DRWE  ; 17    OUT     DR2 write enable.
 /DRRD  ; 18    OUT     DR2 read enable.
  NC2   ; 19    OUT     No function.
  VCC   ; 20    VCC     VCC
; --------------------------------------------------------------------------
```

```
EQUATIONS

  STAT  = /A12 *  A13 *  A14 * /A2 * /A1 *  GRP * IOR   ; W 6.0

  DAC   = /A12 *  A13 *  A14 *  A2 * /A1 *  GRP * IOW   ; W 6.4

  CTC   = /A12 * /A13 *  A14 *              GRP         ; W 4.X

  SIO   =  A12 * /A13 *  A14 *              GRP         ; W 5.X

  DRWE  = /A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 0.2
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOW   ; W 0.4
        + /A12 * /A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 0.6
        +  A12 * /A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 1.0
        +  A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 1.2
        + /A12 *  A13 * /A14 *              GRP * IOW   ; W 2.X
        +  A12 *  A13 * /A14 *              GRP * IOW   ; W 3.X

  DRRD  =  A12 *  A13 * /A14 *              GRP * IOR   ; W 3.X

  W16   = /A12 * /A13 * /A14 * /A2 * /A1 *  GRP         ; RW 0.0
        + /A12 * /A13 * /A14 *  A2 * /A1 *  GRP         ; RW 0.4
        + /A12 * /A13 * /A14 * /A2 *  A1 *  GRP * IOR   ;  R 0.2
        +  A12 * /A13 * /A14 * /A2 *  A1 *  GRP         ; RW 1.2
        + /A12 *  A13 * /A14 *              GRP         ; RW 2.X
        +  A12 *  A13 * /A14 * /A2 *  A1 *  GRP         ; RW 3.2
        +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP         ; RW 3.6
        +  A12 *  A13 * /A14 *  A2 * /A1 *  GRP * IOR   ;  R 3.4
```

## U11 Pattern 0324

```
Title    PT2 Timing Generator
Pattern  0324-3
Revision 3.0

CHIP DR2 PAL16L8                ; Production unit in GAL16V8

; Provides timing for extended cycle to write to DR2 card.

; Name    Pin   Dir     Function
; -----------------------------------------------------------------------
 /IN    ; 1     IN      Go signal. Rising edge initiates a DR2 write cycle.
 /T1    ; 2     IN      Delay line tap 1
 /T2    ; 3     IN      Delay line tap 2
 /T3    ; 4     IN      Delay line tap 3
 /T4    ; 5     IN      Delay line tap 4
 /T5    ; 6     IN      Delay line tap 5
  SPB0  ; 7     IN      Future board functionality.
  SPB1  ; 8     IN      Future board functionality.
  UD    ; 9     IN      Future board functionality.
  GND   ; 10    GND     GND
 /DRRD  ; 11    IN      DR2 read cycle.
 /FIN   ; 12    OUT     Clear cycle flip-flop.
 /PC    ; 13    OUT     Port C output bus enable.
 /PB    ; 14    OUT     Port B output bus enable.
 /PA    ; 15    OUT     Port A output bus enable.
 /PCK   ; 16    OUT     DR2 ck pulse.
 /BIE   ; 17    OUT     DR2 input data enable.
  BDIR  ; 18    OUT     DR2 245 direction control.
 /BEN   ; 19    OUT     DR2 245 enable control.
  VCC   ; 20    VCC     VCC
; -------------------------------------------------------------------------
EQUATIONS

  FIN   =  T1 *  T5

  PC    =  IN
        +  T3
        +  T5
        + /SPB0 *  SPB1 * /UD           ; SELFTEST LOOPBACK

  PB    =  IN
        +  T3
        +  T5
        + /SPB0 *  SPB1 * /UD

  PA    =  IN
        +  T3
        +  T5
        + /SPB0 *  SPB1 * /UD

  BDIR  =  IN +  T1 +  T2 +  T3 +  T4 +  T5

  BEN   =  T1 *  SPB0                   ; T1 + T4 UNLESS LOOPBACK
        +  T1 * /SPB1
        +  T1 *  UD
        +  T4 *  SPB0
        +  T4 * /SPB1
        +  T4 *  UD
        +  DRRD

  BIE   =  DRRD

  PCK   =  T4 *  T1
```

## U31 Pattern 0325

```
Title    PT2 SIO INTERFACE PAL
Pattern  0325-4
Revision 4.0

CHIP SIO PAL16R4                ; Production unit in GAL16V8

; Divides Xtal osc to provide CTC ck
; and provides a 250nS wait state in response to WS signal.

; Name    Pin   Dir     Function
; -----------------------------------------------------------------------
  CK     ; 1     IN      8Mhz oscillator, async to PC bus ck.
 /RD     ; 2     IN      IOR strobe.
 /WR     ; 3     IN      IOW strobe.
 /SIO    ; 4     IN      SIO address valid.
 /CTC    ; 5     IN      CTC address valid.
 /WS     ; 6     IN      Wait-state requested by address decodes.
  NC1    ; 7     IN      No function.
  NC2    ; 8     IN      No function.
  FF     ; 9     IN      Wait-state flip-flop output.
  GND    ; 10    GND     GND
 /OE     ; 11    IN      Output enable.
  WAIT   ; 12    OUT     PC IO_CH_RDY signal.
 /CLR    ; 13    OUT     Clear wait-state flip-flop.
 /QA     ; 14    OUT     No-connect.     Used for ck divider.
 /QD     ; 15    OUT     No-connect.     Used for wait states.
 /QC     ; 16    OUT     No-connect.     Used for wait states.
 /QB     ; 17    OUT     Divide by 4 ck for CTC timers.
  NC3    ; 18    OUT     No function.
  TRG    ; 19    OUT     Set wait-state flip-flop.
  VCC    ; 20    VCC     VCC
; -----------------------------------------------------------------------

EQUATIONS

  QA    :=  RD *  WR                                    ;Clr if RD and WR
        +  /QA                                          ;Count

  QB    :=  RD *  WR                                    ;Clr if RD and WR
        +  /QB * /QA                                    ;Count
        +   QB *  QA                                    ;Hold

  QC    := /FF                                          ;Reset
        +  /QC                                          ;Count

  QD    := /FF                                          ;Reset
        +  /QD * /QC                                    ;Count
        +   QD *  QC                                    ;Hold

  TRG   =   WS

  CLR   =  /QC * /QD                                    ;2 = 1 Wait-state

  WAIT  =   FF                                          ;Pass through
```

## U33 Pattern 0326

```
Title    PT2 SIO MULTIPLEX PAL
Pattern  0326-4
Revision 4.0

;Revised 2/02/94
; Added SMODE to TDA and TDA equations to disable handshake when SMODE high.


CHIP SIO PAL16L8                ; Production unit in GAL16V8

; Name    Pin   Dir     Function
; -----------------------------------------------------------------------
  TXDA  ; 1     IN      Transmit data from SIO ch A.
  TXDB  ; 2     IN      Transmit data from SIO ch B.
  SPB0  ; 3     IN      Invert TXD/RXD ch A.  0=normal, 1=invert.
  SPB1  ; 4     IN      Invert TXD/RXD ch B.  0=normal, 1=invert.
  AACK  ; 5     IN      Port C Multiplexed input data - RXDA or ACKA
  BACK  ; 6     IN      Port C Multiplexed input data - RXDB or ACKB
  RDYA  ; 7     IN      Handshake flip-flop A.
  RDYB  ; 8     IN      Handshake flip-flop B.
  R2    ; 9     IN      Direct RS232 RXDA.
  GND   ; 10    GND     GND
  R1    ; 11    IN      Direct RS232 RXDB.
  RXDA  ; 12    OUT     Receive data to SIO ch A.
  RXDB  ; 13    OUT     Receive data to SIO ch B.
  SMODE ; 14    IN      Select serial mode - 0=DR, 1=direct RS232
  PCSL  ; 15    IN      Port C mode select - 0=ACK, 1=RXD
  NC1   ; 16    XX      No function.
  NC2   ; 17    XX      No function.
  TDB   ; 18    OUT     Port C Multiplexed output data - TXDA or RDYA
  TDA   ; 19    OUT     Port C Multiplexed output data - TXDB or RDYB
  VCC   ; 20    VCC     VCC
; -----------------------------------------------------------------------

EQUATIONS

  TDA   =  TXDA *  PCSL * /SPB0                         ;TXD select to DR
        + /TXDA *  PCSL *  SPB0                         ;Inv TXD select to DR
        +  RDYA * /PCSL * /SMODE                        ;RDY select to DR

  TDB   =  TXDB *  PCSL * /SPB1                         ;TXD select to DR
        + /TXDB *  PCSL *  SPB1                         ;Inv TXD select to DR
        +  RDYB * /PCSL * /SMODE                        ;RDY select to DR

  RXDA  = /AACK * /SMODE * /SPB0                        ;DR serial data
        +  AACK * /SMODE *  SPB0                        ;INV DR Serial data
        +  R1   *  SMODE * /SPB0                        ;RS232 data
        + /R1   *  SMODE *  SPB0                        ;INV RS232 data

  RXDB  = /BACK * /SMODE * /SPB1                        ;DR serial data
        +  BACK * /SMODE *  SPB1                        ;INV DR Serial data
        +  R2   *  SMODE * /SPB1                        ;RS232 data
        + /R2   *  SMODE *  SPB1                        ;INV RS232 data
```

## U6 Pattern 0327

```
Title    PT2 Address Decode PAL
Pattern  0327-1
Revision 2.0


CHIP ADDR5 PAL16L8              ; Production unit in GAL16V8

; Provides misc address decodes.

; Name    Pin   Dir     Function
; -----------------------------------------------------------------------
  A2    ; 1     IN      Address 2
  A1    ; 2     IN      Address 1
  A12   ; 3     IN      Address 12
  A13   ; 4     IN      Address 13
  A14   ; 5     IN      Address 14
 /IOW   ; 6     IN      IOW
 /IOR   ; 7     IN      IOR
  RSTIN ; 8     IN      Reset Input.
  PCK   ; 9     IN      DR2 ck signal.
  GND   ; 10    GND     GND
 /GRP   ; 11    IN      Card enable signal.
 /PTEN  ; 12    OUT     Master PT2 enable signal to DR2 card.
  PPT   ; 13    IN      Input from misc register.
 /RST   ; 14    OUT     Reset board signal.
 /NC2   ; 15    OUT     No function.
 /RCKB  ; 16    OUT     Set RDY latch channel A.
 /RCKA  ; 17    OUT     Set RDY latch channel B.
 /RSTB  ; 18    OUT     Reset RDY latch channel B.
 /RSTA  ; 19    OUT     Reset RDY latch channel A.
  VCC   ; 20    VCC     VCC
; -----------------------------------------------------------------------


EQUATIONS

  PTEN   =  PPT                                         ; Inverter

  RSTIN = /A12 *  A13 *  A14 * /A2 *  A1 *  GRP * IOW   ; W 6.2
        +  RSTIN

  RCKA   =  A12 *  A13 * /A14 * /A2 * /A1 *  GRP * IOW   ; W 3.0
         +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 3.6

  RCKB   =  A12 *  A13 * /A14 * /A2 *  A1 *  GRP * IOW   ; W 3.2
         +  A12 *  A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 3.6

  RSTA   =  A12 * /A13 * /A14 *  A2 * /A1 *  GRP * IOW   ; W 1.4

  RSTB   =  A12 * /A13 * /A14 *  A2 *  A1 *  GRP * IOW   ; W 1.6
```

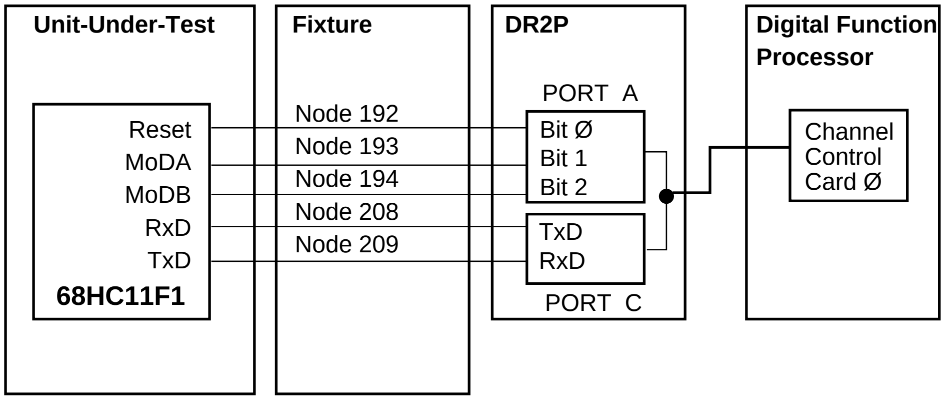
# Custom Example—Serial Boot 4

The 18XX DigFuncProc testsheet sends the subdirectory path and arguments (ALPHA0) to slave.exe on the DF) computer to start the appropriate ptprog.exe. The ptprog.exe program loads a boot program (the .img file) into the 68HC11 RAM via the 68HC11 serial bootloader mode, passing the program to the 68HC11 serially by way of TXDA. Also a serial number (18XX ALPHA0), date, and time is passed to the 6811 at the end of the bootloader program.

The boot program, running internally from the 68HC11 RAM, directs the 68HC11 to first erase ten locations of EEPROM, then load the ten bytes of customer data (the serial number, date, and time) to the EEPROM. The 68HC11 boot program passes back a PASS/FAIL indication to ptprog.exe, which passes the PASS/FAIL information to the 18XX DigFuncProc testsheet.

Ptprog.exe can also be started from the DFP keyboard.

Figure 4.1 Serial Boot Interconnect Diagram



## PT2.INI File—68C11F1

Below is the pt2.ini file used in this example.

```
L,IC1,68HC11F1,eeprom.obj,82,0,0,0
R,eeprom.obj is the formated (compiled) eeprom.asm bootload file
R,format 82 = motorola type S record (eeprom.obj)
```

## PTPROG.C File—68C11F1

Below is the ptprog.c file that boots the 68HC11F1.

```
/*
 ***  Serial Boot Example   ***

Custom Application For Serial Bootload 68HC11F1 - to Program 10 bytes
    of imbedded (internal) EEPROM.

Filename: ptprog.c
Component/s: 68HC11F1

Aliases        : M68HC11F1, MC68HC11F1

Device Manufacturer: Motorola
Device Function: Microcontroller
IC Package: 68 pin plcc
Original Source: based on DFP A.3 - serial bootload example
Fixture Requirements: see Wiring below
Files Required: ptprog.exe (compiled ptprog.c)
                 pt2.ini    (see pt2.ini below)
                 data file  (see pt2.ini below)
Written with OS: DFP  - B.0
                 18xx - F.0
                 To use this program with earlier versions
                     (either DFP or 18xx)
                     comment out the 'send18xxMsg()'
                     function call in the
                     print_Message() function.
Test Description:

   68HC11F1 Microcontroller:8K ROM
                           256 Bytes RAM
                           512 Bytes EEPROM -
                                 Byte/Bulk Erase

   The ptprog program loads a boot program into the
   68HC11 RAM via the serial bootloader mode, passing
   the program to the 68HC11 serially by way of TXDA.
   The boot program, runing internally from the 68HC11
   RAM, directs the 68HC11 to first erase 10 locations
   of EEPROM, and then subsequently load 10 bytes of
   customer data to the EEPROM.  The 68HC11 passes back
   PASS/FAIL information and then proceeds to the monitor
   routine which allows the user (in debug mode) to check
   the contents of the 68HC11 memory (refer to monitor
   function).

   note:
   The data is loaded into the default EEPROM addresses
   starting at 0x0E00.  However, in serial boot mode the
   address for EEPROM starts at 0xFE00, hence the eeprom.asm
   base address calls for 0xFE00, not 0x0E00.  When the
   boot program is finished and the 68HC11 is reset
   the EEPROM base address will then be seen at 0x0E00
   as expected (default address configuration for EEPROM).
   Please note that when using the monitor mode of this
   program, the chip has not yet been reset so the
   addresses to check for EERPOM data start with 0xFE00.

Program Organization Outline:
   1. open com ports and set up port parameters
   2. get the arguments:
         expects a 5 digit serial number
   4. open files:1. pt2.ini -> retrieve:
                         device type
                         byte position
                         .img filename
               2. eeprom.img (data file)
   5. store bootloader program (from datafile - see pt2.ini below)
   6. store eeprom data:5 byte serial number
                  1 byte Year
                  1 byte Month
                  1 byte Day
                  1 byte Hour
                  1 byte Minute
   7. set up CCC/DR2P cards
   8. load bootloader data to 68CH11
         .1.  Ptprog resets 68HC11 into serial boot-loader
               mode.
         .2.  Ptprog passes to the 68HC11, one byte at a time,
               the boot program with the variable data
               (serial number) tacked on to the end.
         .3.  Ptprog delays apx. 5ms for 68HC11 to realize it
               is time to run itself from the program
               just loaded into its RAM. (refer eeprom.asm)
         .4.  Ptprog sends the checksum to the 68HC11.
               checksum = total program bytes plus 10 variable
                           data bytes.
         .5.  68HC11 checks its receive buffer for the ptprog
               checksum, and responds by sending the checksum
               it also tallied to ptprog. (refer eeprom.asm)
         .6.  Both programs -- 68HC11 and ptprog -- compare
               the two checksums.  If either finds a mismatch,
               the program will fail.
              Note: ptprog will time out if 68HC11 does
```

```
                    NOT send a checksum and the program
                    will fail.
        .7. This step will be repeated up to 3 times to
              try to successfully boot load the 68HC11.

   9. Meanwhile - 68HC11 does its 'thing': (refer the eepom.asm)

        .1.  If 68HC11 compare checksums match, it proceeds to
              erase the 10 address locations in EEPROM
              for the variable data, else it proceeds
              to FAIL.
        .2.  After 68HC11 erases the eeprom locations, it
              loads the addresses with the variable data
              After each eeprom byte write, the 68HC11
              will read back and verify the write.
              If any byte does not verify, the program
              will proceed to FAIL.
        .9. If 68HC11 proceeds to FAIL at any time, it
              complements the response regiseter (0x00->0xFF).
        .10. The 68HC11 leaves the configure register
              in the default mode. (EEPROM = 0x0E00,
              NOSEC=1, NCOP=1,ROMON=0,EEON=1)
        .11. The 68HC11 reprotects the CONFIG and EEPROM.
        .12. The 68HC11 waits for a request for the
              PASS/FAIL response.

   10. Ptprog requests the PASS/FAIL status from 68HC11

        .1. ptprog requests the 68HC11 to send
              the response PASS/FAIL.
        .2. 68HC11 checks its recieve buffer and
              sends the response register data.
        .3. ptprog checks the response data,
              0x00 = PASS, 0XFF = FAIL.
            Note: ptprog will time out if 68HC11 does
                    not send the response data.
        .4. 68HC11 goes into monitor mode - waiting
              to receive address locations from
              ptprog if ptprog enters the monitor function.

   11. ptprog cleanup:
        .2. sends PASS/FAIL to 18xx
        .3. checks for monitorFlag
        .4. enters monitor mode if monitorFlag true (only if
                    program is started from
                    DFP keyboard for debug)
        .5. release DR's, close all files + ports
        .6. exit(0)

18xx Worksheet/Argument usage:
   1. Set up 18xx test to take in a 5-digit serial number
        earlier in the 18xx program and store
        in ALPHA0.
   2. DigFuncProc worksheet example:
              Source Dir:mod1
              Arguments:%ALPHA0
              Timeout:5

        mod1    = subdirectory to find ptprog.exe
        %ALPHA0 = serial number to pass to ptprog.exe
        5     = timeout in seconds for DigFuncProc worksheet

Sample pt2.ini-68HC11F1:
   L,IC1,68HC11F1,eeprom.obj,82,0,0,1

        L           = local device tag
        IC1         = board identifier
        68HC11F1= device type
        eeprom.obj= data source file
        82          = format of data source file
                          (82 = motorola s-record)
        0           = memory size
                           (0=default to data file size)
        0           = byte position
        1           = fill character

Wiring         :
   signal:     DFP card:DFP node/s:
   -------     --------------------
   Reset       card 0      192   (Port A)
   ModA        card 0      193   (Port A)
   ModB        card 0      194   (Port A)
   RxD         card 0      208   (Port C - DFP -TxD)
   TxD         card 0      209   (Port C - DFP -RxD)

Statistics-68HC11F1:
   18xx:       O.S. Version - E.3
   DFP:        O.S. Version - A.3
              AmiBios - ISA BUS CLOCK SELECTION = AUTO
              AmiBios - DRAM = FAST
              SmartDrv - Set (times taken after 1 initial
                     run to load smart drv memory)
   Compiler:Turbo C - 3.0
   sample of: 1
   aprox. times for:
        erasing and programming 10 bytes EEPROM in 68HC11F1:
                                 ------
              10 bytes1.20 sec

   NOTE: All times have 200ms added for 18xx memory swapping.
   NOTE: time may be reduced if:
              1.  You have assembler program
        check for erased condition of eeprom rather
        than automatically erase the eeprom address.  This
        programmer did not have time to pursue this, but
        it would make sense if expecting to program
        mainly new devices.
```

```
               2.  You do not need to remap or reconfigure
         the CONFIG reg.  Refer to the REMAP section of
         the eeprom.asm.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <dos.h>
#include <errno.h>
#include <ctype.h>
#include "pt2.h"

#define MATCH0
#define NO_MATCH(-1)
#define NO_REPLY(-2)
#define MAX_WIDTH1// One device and datafile to
                     // be identified via the pt2.ini file
#define BOOTSIZE1024//max size of boot data array
#define EE_DATASIZE10//size of eeprom data array

//Define CCC+DR2P cards/control
//note: bus or channel direction is from the DFP's point of view
#define CD0          //dfp card number
#define CD_ADDRCARD0 //dfp card address (refer pt2.h)

//define dfp port a+b= rst_bus = device modb, moda, rst
#define RST_BUSCD_ADDR|PA_DATA//dfp port a = rst bus
#define RST_BUS_DIRCD_ADDR|PAB_CNTL//dfp port a+b direction

//define dfp port c = ser_bus = device/dfp rxda, txda
#define SER_BUSCD_ADDR|PC_DATA//dfp port c = ser_bus
#define SER_BUS_DIRCD_ADDR|PC_CNTL//dfp port c direction
#define SER_ADAT        CD_ADDR|SIO_ADAT//dfp serial xmit or rec

static charnotice[]={"Digital Function Processor system"
                 " Copyright (C) 1993,1994 Teradyne Inc.\n"},
         filename[MAX_WIDTH][13],
         irec[I_REC_SIZ],//ini record buffer
         msg[I_REC_SIZ],//message buffer
         args[MAX_COM_SIZ],//argument string
         **parg;           //pointer to arguments

static FILE*ini_file,//pointer to pt2.ini file
         *failfile,  //pointer to failfile
         *datafile[MAX_WIDTH];//pointer to data file/s
```

```
static inthandle,    //handle for com port
         bootcount,  //bytecount for bootload file
         monitorFlag;

static unsignedintfailval;//pass-fail value

static unsigned charbootdata[BOOTSIZE],//store bootload data
               eedata[EE_DATASIZE],//store eeprom data
               xmitsum,    //xmit checksum
               recsum;           //receive checksum


//function prototypes
static void open_com_port(void);
static void get_args (int argc, char **argv, char *args);
static void usage(void);
static void open_pt2ini(void);
static int read_ini_data(char *record);
static void open_data_file(void);
static void store_boot_data (void);
static int store_eeprom_data(void);
static void init_dfp_cards(void);
static void initSIO(int card);
static int do_bootload(void);
static char checksum(int port);
static int check_write(void);
static void monitor(void);
static void cleanup(void);
static void print_Message(char *msg);//print msg to BOTH DFP+18xx screens

/**********************************************************/
int
main(int argc, char **argv)
{
  clock_t clock_ticks;//structure to hold elapsed time
  int error;//error variable

  failval=0;//initialize failval = 0x0;

/********************************************************/
/* Open DFP COM port + set port parameters             */
/********************************************************/

  open_com_port();

/********************************************************/
/* Display copyright notice                            */
/********************************************************/

  printf("%s\n",notice);

/********************************************************/
/* Create argument string                              */
/********************************************************/
```

```
  get_args (argc, argv, args);
  printf("The argument string = '%s'.\n",args);

/*********************************************************/
/* Open, read, close pt2.ini*/
/*********************************************************/

  open_pt2ini();

/*********************************************************/
/* Open bootloader data file                             */
/*********************************************************/

  open_data_file();

/*********************************************************/
/* Store Boot loader data                                */
/*********************************************************/

 store_boot_data ();

/*********************************************************/
/* Store EEPROM data                                    */
/*********************************************************/

  error = store_eeprom_data();
  if (error)
    failval = 1;

/*********************************************************/
/* Initialize Channel Control Cards + DR2P Cards         */
/*********************************************************/

  if (!failval)
    init_dfp_cards();

/*********************************************************/
/* Write boot-loader program to processor                */
/*********************************************************/

  if (!failval)
    error = do_bootload();
  if (error)
    failval = 1;

/*********************************************************/
/* Query DUT for successful EEPROM WRITE                 */
/*********************************************************/

  if (!failval)
    error=check_write();
  if (error)
    failval = 1;
```

```
/*******************************************************/
/* Cleanup:                                            */
/* send 18xx PASS/FAIL     */
/* enter monitor mode if flag is set */
/* cleanup() function: release (CLR) DR2p's */
/*                close com handle */
/*                close any open files */
/*******************************************************/

  if (failval)
    {
    printf("FAILED!!\n");
    keep_alive(handle,FAIL);
    }
  else
    {
    printf("PASSED!!\n");
    keep_alive(handle,PASS);
    }

#ifdef TURBO
  clock_ticks = clock();
  printf("Elapsed time = %f seconds.\n",clock_ticks / CLK_TCK);
#endif

  if (monitorFlag)
    {
    monitor();
    }

  cleanup();// release DFP control, close files

  return(0);
}   //end main

/*******************************************************/
/*                PROGRAM FUNCTIONS                    */
/*******************************************************/

//++++++++++ open_com_port function ++++++++++
//Function:  Open the DFP computor com port and set paremters.
//Pre:   No preconditions
//Post:  DPF computer com port open.
//       port parameters: 57600 baud
//                        no parity
//                        8 bits
//                        1 stop bit
//++++++++++++++++++++++++++++++++++++++++++++++
void
open_com_port (void)
{
  //open com port
  if ((handle = open("com1",(int)(O_BINARY+O_RDWR))) == -1)
```

```
    {
    printf("Error opening COM port\n");//not send to 18xx
                           //port not open!
    exit(PTPROG_PRT_ERROR);
    }

  //set up parameters for com port
  if (!serial_set(handle,BAUD57600,PARITY_NONE,LENGTH_8,STOPBIT_1,
             PROT_NONE,0,0))
    {
    printf("Error setting port parameters\n");//not send to 18xx
                                 //port parameters not ok!
    close(handle);
    exit(PTPROG_PARAM_ERROR);
    }
  return;
}

//++++++++++ get_args function ++++++++++
//Function:  Get the optional argument string for ptprog.c.
//
//  This functions seperates the "starting switch"
//  statements for how the program was started from the
//  optional arguments for the program.
//
//  Two possibilites for starting ptprog are
//       1. slave.
//       2. DFP keyboard.
//
//  Slave ALWAYS inserts -s into the argument string to
// alert ptprog that slave initiated the program -- and
// therefore the REST OF THE ARGUMENTS are passed
// following the -s (seperated by space/s).
//
//  For debug when you are starting ptprog from the
// DFP keyboard, you may want to sometimes have the
// arguments come from the keyboard, or sometimes
// from the port (18xx testsheet).
//
// The possible "start switch" combinations:
//       1. ptprog started by slave:
//           ptprog -s (args from slave)
//       2. ptprog started from DFP keyboard:
//           ptprog -s (args from keyboard --emulating slave)
//           ptprog -k (agrs from keyboard)
//           ptprog -km(args keyboard)
//           ptprog -p (args from port --18xx testsheet)
//           ptprog -pm(args from port --18xx testsheet)
//
// (switch meanings: s=slave, k=keyboard, p=port, m=monitor)
//
// Examples with arguments:
//       1. start by slave
//               ptprog -s [arg1] [arg2]
//       2. start from DFP keyboard - type:
//                ptprog -s [arg1] [arg2] <enter>
//                ptprog -k [arg1] [arg2] <enter>
//                ptprog -km [ar1g] [arg2] <enter>
//                ptprog -p  <enter>
//                    18xx testsheet <press start>
//                           [arg1] [arg2]
//                ptprog -pm <enter>
//                    18xx testsheet <press start>
//                           [arg1] [arg2]
//
//  This function will fill the variable args[] with only
// the arguments following the switches for where the program
// was started: the -s, -k, -km, -p, -pm are NOT saved.
// Furthermore, the 'P' command and subdirectory name are NOT
// saved from the port string (18xx string).
//        Example: ptprog -km hello world
//             -> args = "hello world"
//        Example: ptprog  -k
//             -> args = ""  (the null string)
//        Example: ptprog  -p
//             18xx sends string:  Pmod1 hello world\n
//             -> args = "hello world"
//
//Pre:   Ptprog is started from the command line or via Slave.
//       Optional arguments are sent from the command line or via
//          the com port.
//
//Post:  Fills the variable args[] with the argument string.
//  Updates global variable monitorFlag.
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
get_args (int argc, char **argv, char *args)
{
  char *arg_ptr;
  int  idx;

  //initialize to default condition
  monitorFlag = 0;

  if (argc < 2 || argv[1][0] != '-')
    usage();
  switch (argv[1][1])
  {
     case 'k': if (argv[1][2] == 'm')
          monitorFlag = 1;
     case 's': *args = 0x0;
          for (idx=2; idx<argc; idx++)
          { if (idx != 2)
            strcat(args," ");
          strcat(args,argv[idx]);
          }
          break;
     case 'p': if (argv[1][2] == 'm')
```

```
          monitorFlag = 1;
          puts("\nWaiting for arguments from 18XX.");
          puts("Press ESC to exit.\n");
          if(read_port(handle,args,0))
          exit(0);
          //Remove charactors up to the first space
          arg_ptr = strstr(args," ");
          if (arg_ptr == NULL)
          *args = 0x0;
          else
          strcpy(args,arg_ptr + 1);
          break;
     default:  usage();
  }
  return;
}

//+++++++++++ usage function  +++++++++++++++++
//Function:  Display the switch statements allowed
// for how to start this program from
// the command line.
//Pre:Ptprog.exe is started from the command line
// or by Slave.
//Post: Displays the usage argument switches for starting
// ptprog.exe.
// +++++++++++++++++++++++++++++++++++++++++++++++
void
usage(void)
{
   printf("\n%s%s%s%s%s%s%s%s%s%s%s",
   "USAGE: ptprog <switch> [arguments]\n",
   "           switch\n",
   "             -s       Emulate input from Slave via DFP keyboard,\n",
   "                      no Monitor option.\n",
   "             -k[m]    Arguments taken from DFP command line,\n",
   "                      optional Monitor.\n",
   "             -p[m]    Wait for arguments from port (18xx Worksheet),\n",
   "                      optional Monitor.\n",
   "             -h       Help\n",
   "             -?       Help\n",
   "       Note: Switch must be lower case!!!\n");
   exit(0);
}


//++++++++++ open_pt2ini.ini function ++++++++++
//Function:  Opens, reads, closes the pt2.ini file.
//      Calls the read_ini_data function to extract
//       specific information from record/s.
//Pre:none
//Post: If pt2.ini file not exist-> program exits.
// Otherwise:
//      1. fills the local variable irec with the ini.
// 2. Stores specific info from record/s via the
//       read_ini_data function.
//++++++++++++++++++++++++++++++++++++++++++++++++
void
open_pt2ini(void)
{
  int error=0;

  //open pt2.ini file
  if ((ini_file = fopen("pt2.ini","rt")) == NULL)
  { sprintf(msg,"Error opening pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }

  // Read all the lines in the pt2.ini file
  while( fgets( irec, I_REC_SIZ - 1, ini_file ) != NULL )
  { if ((error = read_ini_data(irec)) != 0)
    { if (error == WIDTH_ERROR)
        sprintf(msg,"Error reading pt2.ini--WIDTH_ERROR");
      else
        sprintf(msg,"Error reading pt2.ini\n");
      print_Message(msg);
      cleanup();
      exit(error);
    }
  }

  //check really at end of file and not file reading error
  if( !feof( ini_file ) )
  { sprintf(msg,"Error reading pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }

  fclose(ini_file);
  return;
}

//++++++++++ read_ini_data function ++++++++++++++
//Function:  A. Looks for 'L' tag record. The L tag
// record had device specific information.
// For this program the function gets the
// device type, and data filename. The data
// filename is processed to drop any name
// extensions and adds the .img extension.
//
//Pre:Recieves a record from the pt2.ini file.
//
//Post: If the record is an L record:
//       1. variable device_type =  device type
//       2. variable last_address =  number of bytes to program
//       4. variable filename[][] = datafile/s with .img extension
```

```
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++
int
read_ini_data(char *record)
{
  charimgfile[13],
   bytepos[2],
   code[I_REC_SIZ],
   device_type[30],
   *ptr;
  intpos,
   error = 0;

  // Process L type entries
  if (toupper(record[0]) == 'L')
  {
    // Get device type and byte position
    get_field(3,device_type,record);
    printf("Device programming = %s\n",device_type);
    get_field(7,bytepos,record);
    pos=atoi(bytepos);

    // Get image filename(s)
    get_field(4,imgfile,record);
    ptr = strstr(imgfile,".");
    if (ptr != NULL)
      *ptr = 0x0;
    strcat(imgfile,".img");
    printf("infile = %s\n",imgfile);
    if (pos >= MAX_WIDTH)
      error=WIDTH_ERROR;
    strcpy(filename[pos],imgfile);
  }

  return(error);
}

//++++++++++ open data files function ++++++++++
//Function:  Opens the .img data file/s containing
//       the data for programing
//       device/s.
//      The file/s are pointed to by
//       a datafile pointer array.
//      This application only uses
//       one datafile pointer.
//Pre:none
//Post: If file/s not exist-> program exits.
// Otherwise:
//      1. datafile pointers -> to file/s
void
open_data_file(void)
{
  datafile[0] = fopen(filename[0],"r+b");
  if (datafile[0] == 0)
  {
    sprintf(msg,"Error opening %s\n",filename[0]);
    print_Message(msg);
    cleanup();
    exit(FOPEN_ERROR);
  }
}

//++++++++++ store_boot_data function ++++++++++
//Function:  Stores the bootloader data
//       in the bootdata[] array.
//      Keeps track of the count for
//       the bootcount variable.
//      The data file is then closed.
//      If there is a problem reading the
//       file, the program exits.
//
//Pre:boot data file has been opened.
//Post:  bootarray[] = boot loader data
//  bootcount   = number of bytes in bootarray[]
//  boot data file is closed.
void
store_boot_data (void)
{
  longn;
  intdata;

  rewind(datafile[0]);

  for (n=0; n<BOOTSIZE; n++) //initialize boot data array to all 0's
    bootdata[n]=0;

  n=0;
  while ((data = fgetc(datafile[0])) != EOF)
  {
    bootdata[n]=(unsigned char)data;
    n++;
  }
  bootcount = n;

  if (!feof(datafile[0]))
  {
    sprintf(msg,"Error reading data file\n");
    print_Message(msg);
    cleanup();
    exit(FREAD_ERROR);
  }

  fclose(datafile[0]);
  return;
}

//++++++++++ store_eeprom_data function ++++++++++
//Function: Store the 'write' data to the
```

```
//       writedat[] array.
//       1. load 5 byte serial number
//       3. load the date and time
//
//Pre:The program arguments are pointed to by parg[].
// (this program only expects one argument-> the serial number)
//Post: The eedata[] array is loaded.
//+++++++++++++++++++++++++++++++++++++++++
int
store_eeprom_data(void)
{
  longn;
  int len=0;
  struct dosdate_t ddate;
  struct dostime_t dtime;

  // The following information will be entered into the EEPROM:
  //     0x0E00-0E04Serial Number (5 bytes)
  //     0x0E05      Year
  //     0x0E06      Month
  //     0x0E07      Day
  //     0x0E08      Hour
  //     0x0E09      Minute
  //

  printf("Storing eeprom data\n");
  //break up arguments into individual strings (pt2 libraray function)
  parg=breakUp(args);

  // initialize eedata string
  for (n=0;n<EE_DATASIZE;n++)
    {
    eedata[n] = 0x00;
    }

  // **  get the serial number **

  //if serial number is not 5 digits -> fail
  len=strlen(parg[0]);
  if (len != 5)
  {
    sprintf(msg,"ERROR -> serial number must be 5 digits long \n");
    print_Message(msg);
    return(ERROR);
  }

  //write serial number to writedat array
  // and check for alpha-numeric input
  for (n=0; n<5; n++)
  {
    if (isalnum(parg[0][n]))
      eedata[n]=(unsigned char) parg[0][n];
    else
    {
      sprintf(msg,"ERROR -> serial number must be a letter or a number\n");
      print_Message(msg);
      return(ERROR);
    }
  }

  _dos_gettime(&dtime); //get current time
  _dos_getdate(&ddate);//get current date

  eedata[n]=(unsigned char) ddate.day;
  n++;
  eedata[n]=(unsigned char) ddate.month;
  n++;
  eedata[n]=(unsigned char) (ddate.year-1900);
  n++;
  eedata[n]=(unsigned char) (dtime.hour);
  n++;
  eedata[n]=(unsigned char) (dtime.minute);

  printf("EEPROM data = ");
  for (n=0;n< EE_DATASIZE;n++)
    printf("%2.2x ",eedata[n]);
  printf("\n");

return(NOERROR);
}

//++++++++++ init_dfp_cards function ++++++++++++++
//Function:  Initialize DFP CCC cards and
//       DR2P cards for the DFP program.
//
//Pre:The appropriate CCC/DR2P cards are installed
// in the DFP and 18xx.
//Post: The CCC + DR2P cards are initialized for
// serial mode to program the 68HC11 via the
// device pins: Reset, ModA, ModB, RxD, and Txd.
//
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++
void
init_dfp_cards(void)
{
  pt2init();//pt2 init
  initCard(CD);//init dfp card
  releaseDR(CD);//clear all relays
  setMode(CD,SERIAL);//DFP serial mode
  ptEnable(CD,ON);//Enable DFP

  outpw(RST_BUS_DIR,0xFF00 );//DFP Port A static output --
                     //bit 0 = 68hc11 Reset
                     //bit 1 = 68hc11 ModA
                     //bit 2 = 68hc11 ModB

  outp(SER_BUS_DIR,0x01 ); //DFP Port C --
```

```
                    //bit 0 = out = DFP TxD -> 68hc11 RxD
                    //bit 1 = in  = DFP RxD -> 68hc11 TxD

  outp(SER_BUS,0x00 );//DR2P - enable DFP serial output (TxD) via U73

  initSIO(CD);   //init serial parameters.
  delay(0); // Calibrate delay

  // Set relays
  outpw(CD_ADDR|GA_INST,D_REED|nodA[0]|SET);  //68hc11 Reset
  outpw(CD_ADDR|GA_INST,D_REED|nodA[1]|SET);  //68hc11 ModA
  outpw(CD_ADDR|GA_INST,D_REED|nodA[2]|SET);  //68hc11 ModB
  outpw(CD_ADDR|GA_INST,D_REED|nodA[16]|SET); //68hc11 RxD (CCC- TxD)
  outpw(CD_ADDR|GA_INST,D_REED|nodA[17]|SET); //68hc11 TxD (CCC- RxD)
  delay(50);//allow relays to close

  return;
}

//++++++++++ initSIO function ++++++++++++++
//Function:  Initialize serial I/O communications
//       via the DFP (CCC cards) 85C30
// See pg. 8.5 of the Z8530 manual, table 8-2, polled asynchronous
// Baud rate based on 8MHZ - on the CCC card
// Baud rate Examples:
//     9600 baud: (8MHZ/32 = (250000/9600baud ) -2 = 26-2 = 24
//     11520 baud: (8MHZ/32 = (250000/11520baud ) -2 = 21-2 = 19
//
//Pre:Receives the CCC card number to initialize
//Post: 85C30 is initialized for serial communications
//
//+++++++++++++++++++++++++++++++++++++++++++++++++
void initSIO(int card)
{
  // "SCC Port A"
  inp( card|SIO_ACOM );   //reset SCC chip's internal ptrs to WR0
  sioComWr(card,SIO_ACOM, 9, 0xc0); // force reset
  sioComWr(card,SIO_ACOM, 4, 0x44); // 16* clk, 1 stop bit no parity
  sioComWr(card,SIO_ACOM, 3, 0xc0); // Rx 8bits, Rx disable for now
  sioComWr(card,SIO_ACOM, 5, 0x60); // Tx 8bits, Tx disable for now
  sioComWr(card,SIO_ACOM, 9, 0x03); // interrupts disabled
  sioComWr(card,SIO_ACOM,10, 0x00); // Misc unused SDLC functions
  sioComWr(card,SIO_ACOM,11, 0x56); // Tx&Rx =BRGout, TRxC=BRGout
  sioComWr(card,SIO_ACOM,12, 30);   // *DECIMAL* 19 is for 11520 baud
  sioComWr(card,SIO_ACOM,13, 0x00); // upper byte of const = 0.
  sioComWr(card,SIO_ACOM,14, 0x06); // BRGin = pclk, BRG off
  sioComWr(card,SIO_ACOM,14, 0x07); // BRG enabled
  sioComWr(card,SIO_ACOM, 3, 0xc1); // RX enabled
  sioComWr(card,SIO_ACOM, 5, 0xe8); // TX enabled
  // "SCC Port B"
  inp( card|SIO_BCOM );   //reset SCC chip's internal ptrs to WR0
  sioComWr(card,SIO_BCOM, 4, 0x44); // 16* clk, 1 stop bit no parity
  sioComWr(card,SIO_BCOM, 3, 0xc0); // Rx 8bits, Rx disable for now
  sioComWr(card,SIO_BCOM, 5, 0x60); // Tx 8bits, Tx disable for now
  sioComWr(card,SIO_BCOM,10, 0x00); // Misc unused SDLC functions
  sioComWr(card,SIO_BCOM,11, 0x56); // Tx&Rx =BRGout, TRxC=BRGout
  sioComWr(card,SIO_ACOM,12, 30);   // *DECIMAL* 19 is for 11520 baud
  sioComWr(card,SIO_BCOM,13, 0x00); // upper byte of const = 0.
  sioComWr(card,SIO_BCOM,14, 0x06); // BRGin = pclk, BRG off
  sioComWr(card,SIO_BCOM,14, 0x07); // BRG enabled
  sioComWr(card,SIO_BCOM, 3, 0xc1); // RX enabled
  sioComWr(card,SIO_BCOM, 5, 0xe8); // TX enabled
}

//+++++++++++ do_boot_load function  +++++++++++++++++
//Function:  Loads the boot data and eeprom write
//       data into the 6811.
//      The boot load is checked via the
//       checksum function.
//Pre:boot data and eeprom data have been
// stored in eearray[].
//Post: return error:
//       0 = no error
//
// ++++++++++++++++++++++++++++++++++++++++++++++
int
do_bootload(void)
{
  longn;
  int tries;
  int error = 0;

  printf("Boot-loading 68HC11\n");

  for (tries = 1 ;  tries <4 ; tries++)
    {
    xmitsum = 0;//initialize xmitsum (checksum) variable

    // activate control bits
    outp( RST_BUS, 0x07);   //reset=1 moda=1 modb=1
    outp( RST_BUS, 0x01);   //reset=1 moda=0 modb=0
    delay(1);

    // reset the board CPU while moda=modb=low --> serial boot mode
    outp( RST_BUS, 0x00);   //reset=0 moda=0 modb=0
    delay(1);
    outp( RST_BUS, 0x01);   //reset=1 moda=0 modb=0
    delay(2);

    sioWr(SER_ADAT,0xff);  //allow DUT CPU to discern our baud rate
    delay(2);

    for (n=0; n<bootcount; n++)
      {
      sioWr(SER_ADAT,bootdata[n]);//transmit byte
      xmitsum = xmitsum + bootdata[n];//add to checksum
      delay(2);
      }
    for (n=0 ; n < EE_DATASIZE; n++)//send the command tail (bar code)
      {
      sioWr(SER_ADAT,eedata[n]);//transmit byte
      xmitsum = xmitsum + eedata[n];//add to checksum
      delay(2);
      }

    //compare checksums
    error=checksum(SER_ADAT);

    if (error == 0) break; //0 = match, no need to retry
    } // end of for (tries...

   if (error==NO_REPLY)
   {
     sprintf(msg,"Failed bootload->no checksum received from 68HC11\n");
     print_Message(msg);
   }
   if (error==NO_MATCH)
   {
     sprintf(msg,"Failed bootload->checksums not match\n");
     print_Message(msg);
     sprintf(msg,"  Transmitted checksum = %x\n",xmitsum);
     print_Message(msg);
     sprintf(msg,"  Received checksum = %x\n", recsum);
     print_Message(msg);
   }

    return error;
}

//+++++++++++ checksum function  +++++++++++++++++
//Function:  compare our checksum with DUT's checksum:
// 1. wait 5 ms to allow 6811 to begin boot loader
//       program and calculate checksum
// 2. clear our receiver
// 2. send our port (card) address
// 3. listen for DUT's checksum or time out
// 4. compare checksums
//Pre:boot loader program loaded into DUT
//Post: return:MATCH    = match
//       NO_MATCH = checksum bad
//         NO_REPLY = no reply (timed out)
// ++++++++++++++++++++++++++++++++++++++++++++++
char
checksum(int port)
{
  unsigned int n; // misc counter

  delay(5);//give 6811 a time to calculate its checksum
  while ( sioRdrf(port) ) sioRd(port); // discard garbage

  sioWr(port,xmitsum); // request checksum and await reply
  delay(2);

  for ( n=0 ; ( ( n<40000 ) && ( !sioRdrf(port) ) ) ; n++ );
  if ( !sioRdrf(port) )
  {
    return NO_REPLY; // no reply
  }
  else // got a reply - was it good or bad?
  {
    recsum = sioRd(port);
    if ( recsum == xmitsum )
      return MATCH; //perfect match
    else
      return NO_MATCH;   // got a reply, but not what we expected.
  }
}

//++++++++++ check_write function ++++++++++++++
//Function:  Requests the 68HC11 PASS/FAIL response.
//
//Pre:The 68HC11 successfully booted.
//Post: returns error variable:
//       FAIL      = 0xFF
//       PASS      = 0x00
//       NO RESPONSE = -1
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++++
int
check_write(void)
{
  longn;
  int error = -1;

  printf("Requesting pass/fail response from DUT\n");

  // Empty input buffer ->read buffer till ready flag goes false
  while (sioRdrf(SER_ADAT))
    sioRd(SER_ADAT);

  // request pass-fail
  sioWr(SER_ADAT,0xFF);

  //1000000 allows for apx 240-250 ms delay for programming
  //eeprom with 6811 running with 8MHZ clock
  //(10 erase = 100ms, + 10 writes = 100ms,
  // + reconfigure CONFIG = 10ms, + some insurance since using
  // a simple counting loop )
  //note:  this loop drops out as soon as the Rdrf flag goes true!
  for ( n=0 ; ( ( n<1000000 ) && ( !sioRdrf(SER_ADAT) ) ) ; n++ );

  if ( sioRdrf(SER_ADAT) )
    error = sioRd(SER_ADAT);//error = 0 = PASS
  else
  {
    sprintf(msg,"No Pass/Fail Response received from 68HC11\n");
    print_Message(msg);
  }

   return error;
}

//++++++++++ monitor function ++++++++++
//Function:  Allow user to view the contents of the 6811
// memory via addressing.  The user enters a 4 digit (hex)
// address and the function sends the address to the
// DUT.  The DUT returns the address plus 16 bytes
// of data, starting at the given address.
// The first line of data is in hex, then
// below each hex byte is the same byte in
// ascii if it is between 0x19 and 0x7f.
//
// NOTE: This is an extremely simple monitor.
// Error tolerance is high, and error handling is therefore nil.
// User cannot backspace if he errs in typing the address. -
// Just hit return or the space bar, and try again.  This
// program will always transmit a well-formed address to the
// DUT, and the DUT will always echo it for confirmation.
//
// NOTE: the actual contents written to 68HC11 will
//       be seen starting at address 0xFE00, not 0xE000,
//       since the 68HC11 hasn't been rebooted yet,
//       at which time the memory is remapped according
//       to the reconfiguration in the eeprom.asm program.
//
//Pre:The bootloader was successful
// The user enters a 4 digit (hex) address.
// The user enters 'X' or 'Q' to quit
//Post: 1. Returns the address plus 16 bytes of data in hex
//    and then in ascii starting at the given address.
// 2. Exits if the user entered 'X' or 'Q'.
//++++++++++++++++++++++++++++++++++++++++
void
monitor(void)
{
  unsigned int t; // misc counting variable
  unsigned int time; // misc counting variable
  unsigned long adr; // the "address" part of the query to DUT
  int count; // counts the bytes returned from DUT
  char dutbyte[20]; // the actual bytes returned from the DUT
  char rcvd[20]; // flags indicating reception.
  char cmd[5];   // keyboard input string
  char done = 0; // a loop control item
  char *endptr;

  while (!done)
  {
    // clear read buffer by reading till ready flag goes false
    while (sioRdrf(SER_ADAT))
```

```
{
  sioRd(SER_ADAT);
}

// initialize the receive array
for (count = 0; count < 20 ; count++)
{
  rcvd[count] = 0;
}

printf("\nDUT Monitor.  Enter X to quit.\n");
printf("->");
scanf("%s",cmd);

//input to upper case
for (count = 0; count < 4 ; count++)
  cmd[count] = (char)toupper(cmd[count]);

cmd[4]=0; //make sure last bit always a null (end of string)

// These are the exit criteria: x, X, q, or Q.
if ( (cmd[0] == 'X') || (cmd[0] == 'Q'))
{
  done = 1;
  printf("\aExiting monitor\n");
  break;
}

//convert ascii string into unsigned long (hex = base 16)
adr = strtoul(cmd, &endptr, 16);

// Send M to tell CPU this is a monitor command and
// to insure synchronization of the address bytes.
sioWr(SER_ADAT,'M');
delay(2);

//xmit starting address to DUT
sioWr(SER_ADAT,(char)((adr >> 8) & 0xff));//xmit MSB byte
delay(2);
sioWr(SER_ADAT,(char)(adr & 0xff));//xmit LSB byte
delay(2);

// wait (but not forever) for a response from the DUT.
// we are expecting 18 bytes- 2 address bytes, 16 data bytes
for (count = 0 ; count < 18 ; count++)
{
  // get one byte per pass thru this loop
  // Check rdrfdut, but give up if 30000 false rdrfs
  time = 0;
  while (sioRdrf(SER_ADAT) == 0 && time++ < 30000);
  if (time < 29998) // i.e. if we did not time out,
  {
    dutbyte[count] = sioRd(SER_ADAT);
    rcvd[count] = 1;
```

```
      }
      else
      {
      break;
      }
    }

    if (count==0)
      printf("No reply from DUT...\n");
    else
    {
      printf("\n");
      //first print the 2 bytes of returned address in hex
      for (t = 0 ; ((rcvd[t] > 0) && (t<2)) ; t++)
        printf("%2.2x ",(unsigned char)dutbyte[t]);

      printf(":-> "); //a bit of format

      //second print the 16 bytes of returned data in hex
      for (t = 2 ; ((rcvd[t] > 0) && (t<18)) ; t++)
        printf("%2.2x ",(unsigned char)dutbyte[t]);

      //third print the 16 bytes of data in ascii
      printf("\n\t  ");
      for (t = 2 ; ((rcvd[t] > 0) && (t<18)) ; t++)
      {
        if ((dutbyte[t] > 0x19) && (dutbyte[t] <0x7f))
          printf(" %c ",dutbyte[t]);
        else
          printf(" . "); // if applicable
      }
      printf("\n");
    }
  } //end while !done
}

//++++++++++ cleanup function ++++++++++++++
//Function:  Release the DR2P cards from
// DFP control, close all files+ handles.
//
//Pre:Program is about to exit (return).
//Post: DFP releases DR2P control, all
// files + handles closed.
//
//++++++++++++++++++++++++++++++++++++++++++++++++++++
void
cleanup(void)
{
  releaseDR(CD);// Disable PrompTest
  close(handle);// Close com port handle
  fcloseall();// Close any open files
  return;
}
```

```
//++++++++++ print Message function ++++++++++++++
//Function:  print (send) failure/error message
//       to both DFP and 18xx screen
//
//Pre:Receives msg buffer with formatted string.
//Post: Prints message to DFP screen.
// Sends message to 18xx to printed to 18xx screen.
//
//++++++++++++++++++++++++++++++++++++++++++++++++++++
void
print_Message(char *msg)
{
   printf("%s",msg); //print to DFP screen
   send18xxMsg(handle,DFP_18XX_MSG,msg); //send to 18xx DigFuncProc
                                 //worksheet
  return;
}
```

## Assembly Source Code—68HC11F1

Below is the assembly source code that is booted into into the 68HC11F1 processor for internal eeprom programming. You can modify this code to suit your processor.

```
;************************************************************************
;*                                                                      *
;*         THIS PROGRAM DEVELOPED FOR 68HC11F1 EEPROM LOADING           *
;*                                                                      *
;************************************************************************

;************************************************************************
;*     SPECIFIC EQUATES FOR PROGRAMING 10 BYTES OF DATA INTO EEPROM:
;*-> EEPROM BASE ADDRESS
;*-> STARTING ADDRESS TO PROGRAM EEPROM
;*-> NUMBER OF BYTES TO PROGRAM (CONSECUTIVE)
;*
;************************************************************************
    .EQU PROMORG  ,  $FE00   ; BASE ADDRESS OF EEPROM
    .EQU DATASTRT ,  $FE00   ; STARTING EEPROM ADDRESS TO PROGRAM
    .EQU DATASIZE ,  10      ; NUMBER OF (CONSECUTIVE) BYTES TO PROGRAM

;************************************************************************
;*
;*10 MS DELAY LOOP EQUATE - FOR PROGRAMING EEPROM
;*
;*2 examples are:
;* EQUATE:    XTAL:        E CLOCK:
;*      .EQU PGM10MS  ,  31450   ;  12.58MHZ / 4 = 3.145
;*      .EQU PGM10MS  ,  20000   ;   8.00MHZ / 4 = 2.000
;*
;************************************************************************
;*
    .EQU PGM10MS  ,  20000   ; 10MS DELAY FOR XTAL = 8MHZ

;************************************************************************
;*                                                                      *
;*          MEMORY MAP EQUATES                                          *
;*                                                                      *
;************************************************************************
    .EQU PROGSTAR ,  $0000            ;START OF EXECUTABLE CODE
    .EQU RAMSIZE  ,  1024             ;RAM SIZE OF PROCESSOR
    .EQU EFFECRAM ,  RAMSIZE-DATASIZE ;SIZE OF USABLE RAM LESS EEPROM DATA
;************************************************************************
;*                                                                      *
;*          GENERAL EQUATES                                             *
;*                                                                      *
;************************************************************************
;*
;*  BIT EQUATES
;*
    .EQU BIT0     ,  $01
    .EQU BIT1     ,  $02
    .EQU BIT2     ,  $04
    .EQU BIT3     ,  $08
    .EQU BIT4     ,  $10
    .EQU BIT5     ,  $20
    .EQU BIT6     ,  $40
    .EQU BIT7     ,  $80
;* --------------------------------------------------------------------------
    .EQU REG      ,  $1000    ;BASE ADDRESS FOR CONTROL REGISTERS
;* --------------------------------------------------------------------------
;*
;*-                      DESCRIPTION               PROCESSOR RESET VALUE
    .EQU PORTA    , $00 ;PORT A DATA REGISTER              b'U0000UUU
;*-                      B0-B7 MAY BE DISCRETE I/O
;*-                      PA0-PA2 MAY BE INPUT CAPTURE
;*-                      PA4-PA7 MAY BE OUTPUT COMPARE
;*-                      PA3 MAY BE INPUT CAPTURE OR OUTPUT COMPARE
;* --------------------------------------------------------------------------
    .EQU DDRA     , $01  ;RESERVED FOR DATA DIRECTION REGISTER PORT A
;* --------------------------------------------------------------------------
    .EQU PORTG    , $02 ;PORT G DATA REGISTER              b'U0000UUU
;*-                      MODE 0 OR BOOT = NORMAL I/O PORT
;*-                      MODE 1 = D7,D6....D0
;* --------------------------------------------------------------------------
    .EQU DDRG     , $03  ;RESERVED FOR DATA DIRECTION REGISTER PORT G
;* --------------------------------------------------------------------------
    .EQU PORTB    , $04 ;PORT B DATA REGISTER              b'00000000
;*-                      MODE 0 OR BOOT OUTPUT ONLY PORT
;*-                      MODE 1 OR TEST A15,A14....A8
;* --------------------------------------------------------------------------
    .EQU PORTF    , $05 ;PORT F DATA REGISTER               MODE DEP.
;*-                      MODE 0 OR BOOT = OUTPUT ONLY PORT
;*-                      MODE 1 = A7,A6....A0
;* --------------------------------------------------------------------------
    .EQU PORTC    , $06 ;PORT C DATA REGISTER               MODE DEP.
;*-                      MODE 0 OR BOOT = NORMAL I/O PORT
;*-                      MODE 1 = D7,D6....D0
;* --------------------------------------------------------------------------
    .EQU DDRC     , $07 ;DATA DIRECTION REGISTER FOR PORT C b'00000000
;* --------------------------------------------------------------------------
    .EQU PORTD    , $08 ;PORT D DATA REGISTER                MODE DEP.
;*-                      B5-B0 I/O AS PER DDRD
;*-                      B7 = STRB, B6 = STRA IF MODE 0
;*-                      B7 = R/W,  B6 = AS   IF MODE 1
;* --------------------------------------------------------------------------
    .EQU DDRD     , $09 ;DATA DIRECTION REGISTER PORT D    b'00000000
;*-                      B5 - B0 ONLY
;* --------------------------------------------------------------------------
    .EQU PORTE    , $0A ;I/O PORT E (INPUT ONLY)           b'UUUUUUUU
;*-                      SHARED BY SERIAL A/D
;* --------------------------------------------------------------------------
    .EQU TCNT     , $0E  ;TIMER COUNTER REGISTER (16-BIT)      $0000
    .EQU TCNTH    , $0E ;TIMER COUNTER REGISTER (MSB)      b'00000000
    .EQU TCNTL    , $0F ;TIMER COUNTER REGISTER (LSB)      b'00000000
;* --------------------------------------------------------------------------
```

```
    .EQU TOC2     , $18  ;OUTPUT COMPARE 2 REGISTER (16-BIT)   $FFFF
    .EQU TOC2H    , $18 ;OUTPUT COMPARE 2 REGISTER (MSB)   b'11111111
    .EQU TOC2L    , $19 ;OUTPUT COMPARE 2 REGISTER (LSB)   b'11111111
    .EQU TCTL1    , $20 ;TIMER CONTROL REGISTER 1          b'00000000
;*
;*-        .  OM(X) OL(X) ACTION TAKEN UPON SUCCESSFUL OUTPUT COMPARE
;*-        .    0     0   DISCONNECT TIMER FROM OUTPUT PIN
;*-        .    0     1   TOGGLE OC(X) OUTPUT
;*-        .    1     0   CLEAR OC(X) OUTPUT
;*-        .    1     1   SET OC(X) OUTPUT
;* ----------------------------------------------------------------------
    .EQU TMSK1    , $22 ;TIMER MASK REGISTER 1             b'00000000
;* ----------------------------------------------------------------------
    .EQU TFLG1    , $23 ;TIMER INTERRUPT FLAG REGISTER     b'00000000
;* ----------------------------------------------------------------------
    .EQU OC2F     , BIT6   ;  OC2 INTERRUPT FLAG
;* ----------------------------------------------------------------------
    .EQU SPCR     , $28 ;SPI CONTROL REGISTER             b'000001UU
;* ----------------------------------------------------------------------
    .EQU SPR0     , BIT0    ; SPI CLOCK PRESCALAR
    .EQU SPR1     , BIT1    ; SPI CLOCK PRESCALAR
;*-                  b'00  E/2
;*-                  b'01  E/4
;*-                  b'10  E/16
;*-                  b'11  E/32
    .EQU CPHA     , BIT2    ; CLOCK PHASE
    .EQU CPOL     , BIT3    ; CLOCK POLARITY
    .EQU MSTR     , BIT4    ; MASTER/SLAVE SELECT (1 = MASTER)
    .EQU DWOM     , BIT5    ; PORT D WIRE-OR MODE
    .EQU SPE      , BIT6    ; SPI ENABLE
    .EQU SPIE     , BIT7    ; SPI INTERRUPT ENABLE
;* ----------------------------------------------------------------------
    .EQU SCSR     , $2E ;SCI STATUS REGISTER              b'11000000
;* ----------------------------------------------------------------------
    .EQU SCDR     , $2F ;SCI DATA REGISTER                b'UUUUUUUU
;*-                    READS OF THIS REGISTER ACCESS THE RECEIVE DATA
;*-                    BUFFER
;*-                    WRITES TO THIS REGISTER ACCESS THE TRANSMIT
;*-                    DATA BUFFER
;* ----------------------------------------------------------------------
    .EQU BPROT    , $35 ;EEPROM BLOCK PROTECT REGISTER    b'UUUUUUUU
;* ----------------------------------------------------------------------
;*-
    .EQU BPRT0    , BIT0    ; PROTECT BLOCK $XE00-$XE1F (1=PROTECT)
    .EQU BPRT1    , BIT1    ; PROTECT BLOCK $XE20-$XE5F (1=PROTECT)
    .EQU BPRT2    , BIT2    ; PROTECT BLOCK $XE60-$XEDF (1=PROTECT)
    .EQU BPRT3    , BIT3    ; PROTECT BLOCK $XEE0-$XFFF (1=PROTECT)
;*-
;*-
    .EQU PTCON    , BIT4    ; 1 = PROTECT CONFIGURATION REGISTER
;* ----------------------------------------------------------------------
    .EQU PPROG    , $3B ;EEPROM PROGRAMMING CONTROL        b'00000000
;* ----------------------------------------------------------------------
    .EQU EEPGM    , BIT0   ;  TURN ON EEPROM PROGRAMMING VOLTAGE
    .EQU EELAT    , BIT1   ;  ENABLE EEPROM TO BE PROGRAMMED OR ERASED
    .EQU ERASE    , BIT2   ;  ERASE EEPROM ENABLE
    .EQU ROW      , BIT3   ;  ERASE ROW/ENTIRE EEPROM
    .EQU BYTE     , BIT4   ;  ERASE BYTE
    .EQU EVEN     , BIT6   ;  PROGRAM EVEN ROWS (TEST)
    .EQU ODD      , BIT7   ;  PROGRAM ODD ROWS (TEST)
;* -------------------------------------------------------------------------
    .EQU HPRIO    , $3C ;HIGHEST PRIORITY I INTERRUPT      b'----0101
;* -------------------------------------------------------------------------
    .EQU PSEL0    , BIT0   ;  CODE TO SELECT INTERRUPT SOURCE
    .EQU PSEL1    , BIT1   ;  "   "   "       "        "
    .EQU PSEL2    , BIT2   ;  "   "   "       "        "
    .EQU PSEL3    , BIT3   ;  "   "   "       "        "
;*-                    b'0000  TIMER OVERFLOW
;*-                    b'0001  PULSE ACCUMULATOR OVERFLOW
;*-                    b'0010  PULSE ACCUMULATOR INPUT EDGE
;*-                    b'0011  SPI SERIAL TRANSFER COMPLETE
;*-                    b'0100  SCI SERIAL TRANSFER COMPLETE
;*-                    b'0101  RESERVED (DEFAULT TO IRQ)
;*-                    b'0110  IRQ (EXTERNAL PIN OR PARALLEL I/O)
;*-                    b'0111  REAL TIME INTERRUPT
;*-                    b'1000  TIMER INPUT CAPTURE 1
;*-                    b'1001  TIMER INPUT CAPTURE 2
;*-                    b'1010  TIMER INPUT CAPTURE 3
;*-                    b'1011  TIMER OUTPUT COMPARE 1
;*-                    b'1100  TIMER OUTPUT COMPARE 2
;*-                    b'1101  TIMER OUTPUT COMPARE 3
;*-                    b'1110  TIMER OUTPUT COMPARE 4
;*-                    b'1111  TIMER OUTPUT COMPARE 5
    .EQU IRV      , BIT4    ; INTERNAL READ VISIBILITY
    .EQU MDA      , BIT5    ; MODE SELECT A
    .EQU SMOD     , BIT6    ; SPECIAL MODE SELECT
    .EQU RBOOT    , BIT7    ; READ BOOTSTRAP ROM
;* -------------------------------------------------------------------------
    .EQU CONFIG   , $3F ;COP, ROM, EEPROM ENABLES          b'UUUUUUUU
;* -------------------------------------------------------------------------
    .EQU EEON     , BIT0   ;  1 = ENABLE EEPROM
    .EQU ROMON    , BIT1   ;  1 = ENABLE ROM (NOT USED IN GMP6)
    .EQU NOCOP    , BIT2   ;  0 = ENABLE COP
    .EQU NOSEC    , BIT3   ;  0 = SECURITY MODE ENABLE (NOT USED IN GMP6)
;*
;* EEPROM MAP POSITION - EE0 THRU EE3 SPECIFY MOST SIGNIFICANT
;* ADDRESS BITS OF EEPROM LOCN.  EEPROM IS LOCATED FROM $XE00-$XFFF
;*
    .EQU EE0      , BIT4   ;  EEPROM MAP POSITION
    .EQU EE1      , BIT5   ;  EEPROM MAP POSITION
    .EQU EE2      , BIT6   ;  EEPROM MAP POSITION
    .EQU EE3      , BIT7   ;  EEPROM MAP POSITION


;*
;**************************************************************************
;*                                                                        *
;*      MAIN    F U N C T I O N A L    P R O G R A M                      *
;*                                                                        *
;*************************************************************************
    .ORG  PROGSTAR
         SEI                ;DISABLE INTERRUPTS
         LDS      #$03FF    ;SET STACK POINTER TO TOP OF RAM
         LDX      #REG      ;LOAD INDEX OFFSET
;*
;* STAY IN SPECIAL BOOTSTRAP MODE, DISABLE BOOT ROM, MAKE CHECKSUM FOR DFP
;*
         LDAA     #SMOD+PSEL2+PSEL0
         STAA     HPRIO,X
         BSET     SPCR,X,DWOM ;PORTD = WIRE "OR" OUTPUTS
         CLR      TCTL1,X     ;DISCONNECT TIMERS FROM OUTPUT PINS
         CLR      TMSK1,X     ;DON'T ALLOW TIMER INTERRUPTS
         CLR      PPROG,X     ;PUT EEPROM IN READ MODE
         JSR      CHECKSUM
;*
;* ERASE BARCODE EEPROM BYTES
;*
         LDAA     #PTCON       ;PROTECT CONFIG, UNPROTECT EEPROM
         STAA     BPROT,X
         LDY      #DATASTRT     ;LOAD BARCODE EEPROM START ADDR INTO Y
DOERASE  LDAA     #BYTE+ERASE+EELAT ;PUT EEPROM IN BYTE ERASE MODE
         STAA     PPROG,X
         STAA     0,Y          ;STORE ANY DATA TO EEPROM ADDR TO BE ERASED
         LDAA     #BYTE+ERASE+EELAT+EEPGM   ;ENABLE EEPROM PROGRAMMING VOLTAGE
         STAA     PPROG,X
         LDD      #PGM10MS     ;DELAY FOR 10 MILLISECONDS
         JSR      DELAY
         CLR      PPROG,X     ;EEPROM IN READ MODE, PROGRAM VOLTAGE OFF
         INY                   ;ADVANCE TO NEXT BARCODE EEPROM ADDR
         CPY      #DATASTRT+DATASIZE
         ;HAS Y INCREMENTED PAST LAST BARCODE ADDR ?
         BNE      DOERASE      ;IF NOT, CONTINUE ERASE, ELSE DROP THROUGH
;*
;* MAIN EEPROM PROGRAMMING ALGORITHM BEGINS HERE
;* NOTE: THIS ROUTINE PROGRAMS 16 BYTES OF DATA FROM RAM TO EEPROM
;*
         LDY      #DATASTRT     ;STORE START BARCODE EEPROM ADDR IN Y
         LDX      #TESEND      ;LOAD ADDR OF FIRST RAM DATA IN X
         PSHX                  ;PUSH ADDR OF FIRST RAM DATA ONTO STACK
EXEC0100 LDX      #REG         ;LOAD INDEX OFFSET
         LDAA     #EELAT       ;SET EEPROM LATCH CONTROL
         STAA     PPROG,X
         PULX                  ;RETRIEVE RAM ADDRESS FROM STACK
         LDAB     0,X          ;PUT DATA STORED IN RAM INTO ACCB
         STAB     0,Y          ;WRITE DATA TO EEPROM
         PSHX                  ;PUSH RAM ADDR ONTO STACK
         LDX      #REG         ;LOAD INDEX OFFSET
         LDAA     #EELAT+EEPGM ;ENABLE EEPROM PROGRAMMING VOLTAGE
         STAA     PPROG,X
         PSHB                  ;PUSH RAM DATA ONTO STACK
         LDD      #PGM10MS     ;DELAY FOR 10 MILLISECONDS
         JSR      DELAY
         PULB                   ;RETRIEVE RAM DATA FROM STACK
         CLR      PPROG,X       ;EEPROM IN READ MODE, PROGRAM VOLTAGE OFF
         LDAA     0,Y           ;READ EEPROM DATA INTO ACCA
         CBA                    ;DATA COMPARE: RAM DATA TO EEPROM
         BNE      FAIL          ;NO MATCH, BRANCH 'FAIL'
         INY                    ;ELSE, POINT TO NEXT EEPROM LOCATION
         CPY      #DATASTRT+DATASIZE
       ;HAS Y INCREMENTED PAST LAST BARCODE ADDR ?
         BEQ      REMAP         ;IF NOT, CONTINUE ERASE, ELSE BRANCH TO REMAP
         PULX                   ;RETRIEVE RAM ADDR FROM STACK
         INX                    ;POINT TO NEXT LOCATION IN RAM
         PSHX                   ;PUSH INCREMENTED RAM ADDR ONTO STACK
         JMP      EXEC0100      ;AND DO NEXT BYTE
;*
;* FAIL: SET RESPONSE REG TO $FF
;* (NOTE: PASS= NO CHANGE TO RESPONSE REG= $00)
;*
FAIL     COM      RESPONSE      ;COMPLEMENT RESPONSE TO BE $FF (FAIL CODE)

;*
;* CHANGE CONFIG EEPROM REG TO ENABLE EEPROM AND REMAP IT TO $0E00
;*   Actually this is the default conditition for the 68HC11A1. The
;*   address for EEPROM can be remapped to any 4K boundries starting
;*   at address $xE00, where x = the 4 upper bits of the CONFIG
;*   register.  Default for the CONFIG register for 68HC11A1 = $0D,
;*   which is what this example is using, and therefore the EEPROM
;*   location is at $0E00.  If your application calls for changing
;*   the starting address for the EEPROM, then the code below will
;*   reconfigure the starting address, as well as allow you to change
;*   the NOSEC, NOCOP, ROMON, EEON bits in the CONFIG register.
;*   If you are using the default parameter of $0D, and the devices
;*   you will be programing never have had their CONFIG register
;*   changed, you could go directly to DONE by changing the above
;*   BEQ from    BEQ REMAP   to    BEQ DONE.
;*

REMAP    LDX      #REG           ;LOAD INDEX OFFSET
         LDAA     #BPRT3+BPRT2+BPRT1+BPRT0  ;UNPROTECT CONFIG, PROTECT EEPROM
         STAA     BPROT,X
         LDAA     #ERASE+EELAT      ;PUT EEPROM IN BULK ERASE MODE
         STAA     PPROG,X
         STAA     CONFIG,X          ;WRITE ANY DATA TO CONFIG ADDRESS
         LDAA     #ERASE+EELAT+EEPGM   ;ENABLE EEPROM PROGRAM VLTG
         STAA     PPROG,X
         LDD      #PGM10MS          ;DELAY FOR 10 MILLISECONDS
         JSR      DELAY
         LDAA     #EELAT            ;DISABLE ERASE MODE AND EEPROM PROGRAM VLTG
         STAA     PPROG,X
         LDAA     #NOSEC+NOCOP+EEON  ;CONFIG DATA = $0D= factory default
     ;ENABLE EEPROM AT $0E00=factory default
         STAA     CONFIG,X          ;WRITE DATA TO CONFIG REG
         LDAA     #EELAT+EEPGM      ;ENABLE EEPROM PROGRAM VLTG
         STAA     PPROG,X
         LDD      #PGM10MS          ;DELAY FOR 10 MILLISECONDS
         JSR      DELAY
         CLR      PPROG,X      ;EEPROM IN READ MODE, PROGRAM VOLTAGE OFF

;*
;* DONE PROGRAMMING SO LEAVE CONFIG AND EEPROM PROTECTED.

DONE     LDAA     #PTCON+BPRT3+BPRT2+BPRT1+BPRT0 ;PROTECT CONFIG AND EEPROM
         STAA     BPROT,X

;*
;* SEND THE RESPONSE (PASS/FAIL)
;*
         BRCLR    SCSR,X,$20,* ;WAIT FOR RDRF TO SET (REQUEST RESPONSE)
         LDAA     SCDR,X       ;GET RECEIVED DATA
         LDAA     RESPONSE     ;LOAD ACCA WITH RESPONSE
         BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
         STAA     SCDR,X       ;TRANSMIT RESPONSE FOR HNDSHK
;*
;* MONITOR: WAIT FOR ADDRESS FROM DFP - RETURN THE THE ADDRESS TO DFP.
;* THEN SEND DFP 16 BYTES DATA STARTING FROM THE ADDRESS.
;* NOTE: EEPROM WILL APPEAR AT $FE00 SINCE 68HC11 WAS BOOTED IN SPECIAL
;* BOOTSTRAP MODE.  ALSO, UPPER FOUR BITS OF CONFIG REG ARE FORCED HIGH
;* IN THIS MODE, AND WILL READ AS AN $F if the CONFIG REG LOCATION
;* IS CHECKED.
;*
EXEC0505 BRCLR    SCSR,X,$20,* ;WAIT FOR RDRF TO SET
         LDAA     SCDR,X       ;GET RECEIVED DATA (COMMAND)
         CMPA     #$4D         ;COMPARE COMMAND TO AN "M"
         BNE      EXEC0505     ;IF NOT AN "M", CONTINUE TO WAIT
         BRCLR    SCSR,X,$20,* ;ELSE, WAIT FOR RDRF TO SET AGAIN
         LDAA     SCDR,X       ;GET RECEIVED DATA (READ ADDRESS HIGH BYTE)
         STAA     READADDH     ;STORE IN RESERVED RAM LOCATION
         BRCLR    SCSR,X,$20,* ;WAIT FOR RDRF TO SET AGAIN
         LDAA     SCDR,X       ;GET RECEIVED DATA (READ ADDRESS LOW BYTE)
         STAA     READADDL     ;STORE IN RESERVED RAM LOCATION
;*
;* XMT STARTING READBACK ADDR TO SERIAL PORT FOR HANDSHAKE
;*
         LDAA    READADDH      ;LOAD MSB OF READ ADDR TO ACCA
         BRCLR   SCSR,X,$80,*  ;WAIT FOR TRANSMIT DATA REG TO EMPTY
         STAA    SCDR,X        ;TRANSMIT BYTE
         LDAA    READADDL      ;LOAD LSB OF READ ADDR TO ACCA
         BRCLR   SCSR,X,$80,*  ;WAIT FOR TRANSMIT DATA REG TO EMPTY
         STAA    SCDR,X        ;TRANSMIT BYTE
;*
;* READ 16 BYTES AND TRANSMIT OUT SERIAL PORT
;*
         LDY     READADDH      ;POINT TO BEGINNING ADDRESS TO READ
         LDAB    #16           ;LOAD NUMBER OF BYTES TO READ
EXEC0600 LDAA    0,Y           ;READ A BYTE
         BRCLR   SCSR,X,$80,*  ;WAIT FOR TRANSMIT DATA REG TO EMPTY
         STAA    SCDR,X        ;TRANSMIT BYTE
         DECB                  ;DECREMENT BYTE COUNT
         BEQ     EXEC0505      ;IF ZERO, RETURN TO SERIAL PORT MONITOR
         INY                   ;ELSE, POINT TO NEXT ADDRESS TO READ
         BRA     EXEC0600      ;CONTINUE READING

;*************************************************************************
;*                                                                       *
;* CHECKSUM: ADD UP DATA BYTES OF PROGRAM STORED IN RAM, TRANSMIT TO DFP*
;*                                                                       *
;*************************************************************************
    .EQU  CHECKSUM, *
         LDY      #TESEND+10 ;LOAD ADDRESS OF PROGRAM END + 16 DATA INTO Y
         CLRA                 ;CLEAR A TO ZERO
ADDLOOP  DEY                  ;DECREMENT Y TO POINT AT NEXT LOWER ADDRESS
         BEQ      LASTADDR    ;IF Y IS ZERO, BRANCH OUT OF LOOP
         ADDA     0,Y         ;ELSE ADD DATA FROM ADDRESS "Y" TO A
         BRA      ADDLOOP     ;CONTINUE LOOP FOR ADDING UP DATA BYTES
LASTADDR ADDA     0,Y         ;ADD IN BYTE AT ADDRESS 0000
         BRCLR    SCSR,X,$20,* ;WAIT FOR RDRF TO SET
         LDAB     SCDR,X       ;GET RECEIVED DATA
         BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
         STAA     SCDR,X       ;TRANSMIT CHECKSUM TO DFP

;         LDAA     #$01
;        BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
;        STAA     SCDR,X      ;TRANSMIT CHECKSUM TO DFP
;        LDAA     #$02
;        BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
;        STAA     SCDR,X      ;TRANSMIT CHECKSUM TO DFP
;        LDAA     #$03
;        BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
;        STAA     SCDR,X      ;TRANSMIT CHECKSUM TO DFP
;        LDAA     #$04
;        BRCLR    SCSR,X,$80,* ;WAIT FOR TRANSMIT DATA REG TO EMPTY
;        STAA     SCDR,X      ;TRANSMIT CHECKSUM TO DFP

         CBA                   ;DO CHECKSUMS MATCH
         BEQ      RETURN
         JMP      FAIL        ;IF NOT, GO BACK AND WAIT FOR ANOTHER BYTE
RETURN   RTS
;*************************************************************************
;*                                                                       *
;*  DELAY: PROVIDES DELAY FOR THE COUNTS IN 'ACCD'                       *
;*                                                                       *
;*************************************************************************
    .EQU  DELAY, *
         ADDD     TCNTH,X   ;ADD FREE RUNNING TIMER PRESENT VALUE
         STD      TOC2H,X   ;SAVE COMPARE TIME
         LDAA     #OC2F
         STAA     TFLG1,X   ;CLEAR ANY SET OUTPUT COMPARE FLAG
         BRCLR    TFLG1,X,OC2F,* ;WAIT FOR OUTPUT COMPARE IN 10 MILLISECONDS
         RTS
RESPONSE     .DB  0  ;PASS/FAIL STATUS REGISTER; $00 IS PASS CODE
READADDH     .DB  0
READADDL     .DB  0
;*************************************************************************
```

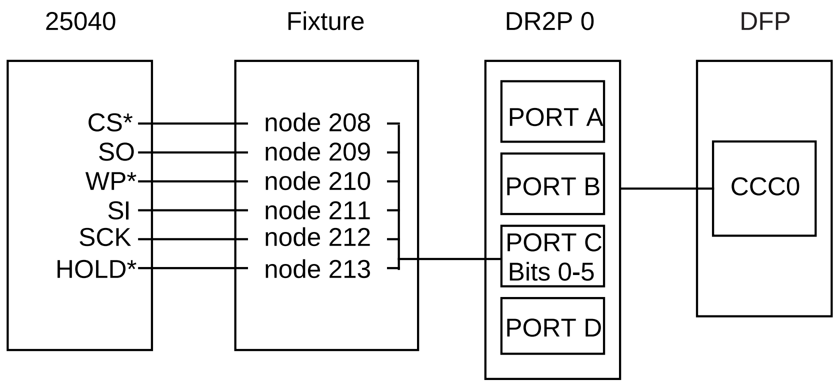
```
;*                                                                      *
;*  DEFINE PROGRAM END LOCATION WHERE DATA FOR EEPROM IS STORED IN RAM  *
;*                                                                      *
;************************************************************************
    .EQU  TESEND, *
         .END
```

# Custom Example — Serial Flash in Free Air 5

The 18XX DigFuncProc testsheet sends the subdirectory path and arguments (ALPHA0) to slave.exe on the DFP computer to start the appropriate ptprog.exe. The ptprog.exe program writes an 8-digit serial number (18XX ALPHA0) and the current date to the EEPROM, filling the rest of the memory with 0xFF. Ptprog.exe then verifies the memory write and sends the 18XX DigFuncProc testsheet a PASS/FAIL indication.

Ptprog.exe can also be started from the DFP keyboard.

Figure 5.1 Serial Flash In Free Air Interconnect Diagram



## PT2.INI File

Below is the pt2.ini file used in this example.

```
L,IC1,25040,,,512,0,
```

## PTPROG.C File

Below is the ptprog.c file used in this example.

```
/*
***  Serial EEPROM Example ***

Custom Application For programming 25040 serial eeprom
Filename : ptprog.c
Component/s: 25040
Aliases        : X25040

Device Manufacturer: Xicor
Device Function: 25040:512 x 8 memory
IC Package: 8 Pin SMD
Written by: Barbara Ryan - Teradyne - ATWC
Date Created: 10-OCT-95
Original Source: 25040,  6-JULY-94
Fixture Requirements: see wiring below
Files Required: ptprog.exe (compiled ptprog.c)
                 pt2.ini    (see below)
Written with OS: DFP -  B.0
                 18XX - F.0
                 To use this program with earlier versions
                     (either DFP or 18xx)
                     comment out the 'send18xxMsg()'
                     function call in the
                     print_Message() function.
Test Description:

   25040 eeprom Memory:4096 bit (512 x 8-Bit)
                     CMOS
                     5.0 Volt-write cycles

   This application writes an 8-digit serial number,
   the current date, and fills the rest of the
   eeprom memory with 0xFF.

   The program will setup the writedat[] array
         with the following data:
         array position:
         0-7    --> 8 digit (alpha-numeric) serial number
         8      --> current day
         9      --> current month
         10     --> current year
         11-512 --> 0xff
   The data is then written to the eeprom, each
         array position corresponds to the
         eeprom address location to write to.

   After the data is written to the eeprom, the
         same addresses are read and stored in
         a the readdat[] array and compared to the
         write array contents, byte by byte for
         verification.

   NOTE: This program was written for an application
         with the pin HOLD* = WP* = pull up.

   NOTE: The monitor routine is very simple and does
         no real error checking.  BE SURE to
         use a valid address in the DUT address
         range.  The valid range is 000-1ff, therefore
         the valid range to get 16 bytes of data
         is 000-1f0.  Typing 1ff, for instance,
         causes the memory to simply roll over,
         and you get byte 1ff and then bytes
         0-14!

Program Organization Outline:
   1. open com ports
   2. set up port parameters
   3. get the arguments
   4. open files:1. pt2.ini -> retrieve
                     device type
                     device size (last_address)
   5. set up CCC/DR2P cards
   6. get serial number and date - fill writedat[] array.
   7. program device
   8. verify device
   9. cleanup:
         write PASS/FAIL to 18xx (pcom.exe)
         enter monitor mode if monitor flag true (only if
               program started from DFP keyboard for
               debug)
         close all files and ports
         release DR's
         exit(0)

18xx Worksheet/Argument usage:
   1. Set up 18xx test to take in an 8-digit serial number
         earlier in the 18xx program and store
         in ALPHA0.

   2. DigFuncProc worksheet example:
               Source Dir:mod1
               Arguments:%ALPHA0
               Timeout:5
         mod1    = subdirectory to find ptprog.exe
         %ALPHA0 = serial number to pass to ptprog.exe
         5     = timeout in seconds for DigFuncProc worksheet

  //note: no keep_alive()s are sent to 18xx to keep the 18xx from
  //timing out since this program typically takes under
  // 2 sec apx. (refer to Statistics below)

Sample pt2.ini-25040:
   L,IC1,25040,,,512,0,

   where:L     = local device tag
         IC1   = board identifier
         25040 = device type
                 = data file =   not used in this program
               = format type = not used in this program
         512   = memory size
         0     = byte position
               = fill character = not used in this program

Wiring         :

   signal:     DFP card:DFP node/s:
   -------     --------------------
   CS*         card 0      208   (Port C)
   SO          card 0      209     (Port C)
   WP*         card 0      210     (Port C) *
   SI          card 0      211     (Port C)
   SCK         card 0      212     (Port C)
   HOLD*       card 0      213     (Port C) *

   * NOTE: this application HOLD* = WP* = pulled high

Statistics-25040:
   18xx:       O.S. Version - E.3/F.0
   DFP:        O.S. Version - A.3/B.0
               AmiBios - ISA BUS CLOCK SELECTION = AUTO
               AmiBios - DRAM = FAST
               SmartDrv - Set (times taken after 1 initial
                     run to load smart drive memroy)
   Compiler:Turbo C - 3.0
   sample of: 1X25040 (Xicor)
   aprox. times for programming a 25040:
                                     ------
   using pagewr:
         4K  (512 x 8-bits)  0.8  sec

   using bytewr:
         4K  (512 x 8-bits)  2.0  sec

   (NOTE: Times have 200ms added for 18xx memory swapping)
*/

#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <dos.h>
#include <errno.h>
#include <ctype.h>
#include "pt2.h"

//define type of write method - page write or byte write
#define PAGE_WRITE1
#define BYTE_WRITE2
#define TYPEPAGE_WRITE

//define CCC/DR2p card used
#define CD0          //DFP ->card number
#define CD_ADDRCARD0 //DFP ->card addr (refer pt2.h)
#define DATA_BUSCD_ADDR|PC_DATA//DFP ->data bus - data
#define DATA_BUS_DIRCD_ADDR|PC_CNTL//DFP ->data bus direction

//define device signals
#define CS_H0x01//CS*   = high
#define CS_L0x00//CS*   = low
#define SO_H0x02//SO    = high
#define SO_L0x00//SO    = low
#define WP_H0x04//WP*   = high
#define WP_L0x00//WP*   = low
#define SI_H0x08//SI    = high
#define SI_L0x00//SI    = low
#define SK_H0x10//SCK   = high
#define SK_L0x00//SCK   = low
#define HD_H0x20//HOLD* = high
#define HD_L0x00//HOLD* = low

static charnotice[]={"Digital Function Processor system"
                " Copyright (C) 1993,1994 Teradyne Inc.\n"},
         args[MAX_COM_SIZ],//argument string
         irec[I_REC_SIZ],  //ini file record buffer
         msg[I_REC_SIZ];  //message buffer

static FILE*ini_file,//pt2.ini file pointer
               *failfile;//fail file pointer
static unsigned charreaddat[512],//store data read from eeprom
               writedat[512];  //store data to write to eeprom
static inthandle,    //com port handle
               type,       //page write or byte write
               monitorFlag;
static longlast_address;
static unsigned intfailval;//failure value
```

```
//general program functions
static void open_com_port (void);
static void get_args (int argc, char **argv, char *args);
static void usage(void);
static void open_pt2ini(void);
static int store_data (void);//store serial #,date,time
static int read_ini_data(char *record);
static void init_dfp_cards(void);
static void monitor(void);
static void cleanup(void);
static void print_Message(char *msg);//print msg to DFP + 18xx screen

//device functions
static void disable_chip (void);
static int program_memory (int type);
static int verify_memory (void);

static int bytewr (int addr, unsigned char wrdat);  //write byte eeprom
static int pagewr (int addr);//write page eeprom

static void write_8_bits (int data);//write 8 bits (MSB to LSB)
static int read_8_bits();//read 8 bits (MSB to LSB)
static void si_low (void);//serial input (SI) low
static void si_high (void);//serial input (SI) high

static void wren (void);//write enable
static void wrdi (void);//write disable
static unsigned char rdsr (void);//read status reg
static void wrsr (void);//write status reg
static void rd (int start_addr, int end_addr);//read eeprom

/**********************************************************/

int main(int argc, char *argv[])
{
  int n,i;
  clock_tclock_ticks;

  failval=0;// initialize failure flag

/********************************************************/
/* Open DFP COM port + set port parameters             */
/********************************************************/

  open_com_port();

/********************************************************/
/* Display copyright notice                             */
/********************************************************/

  printf("%s\n",notice);

/********************************************************/
/* Create argument string                               */
```

```
/********************************************************/

  get_args (argc, argv, args);
  printf("The argument string = '%s'.\n",args);

/********************************************************/
/* Open, read, close pt2.ini*/
/********************************************************/

  open_pt2ini();

/********************************************************/
/* Initialize Channel Control Cards + DR2P Cards        */
/********************************************************/

  init_dfp_cards();

/********************************************************/
/* Initialize Device                                */
/********************************************************/

  disable_chip();

/*******************************************************/
/* store data - serial number, date                    */
/*******************************************************/

  if (store_data())
  { sprintf(msg,"Error - Serial # must be 8 digits or letters\n");
    print_Message(msg);
    failval=1;
  }

/********************************************************/
/* Program eeprom                                       */
/********************************************************/

  if (!failval)
    if (program_memory(TYPE))
    { sprintf(msg,"Error - eeprom failed to program\n");
      print_Message(msg);
      failval=1;
    }

//********************************************************/
//* Verify eeprom                                        */
//********************************************************/

  if (!failval)
    if (verify_memory())
    { sprintf(msg,"Error - eeprom failed to verify\n");
      print_Message(msg);
      failval=1;
    }
```

```
/*******************************************************/
/* Cleanup:                                            */
/* send 18xx PASS/FAIL     */
/* enter monitor mode if flag is set */
/* cleanup() function: release (CLR) DR2p's */
/*                close com handle */
/*                close any open files */
/*******************************************************/

  if (failval)
  { printf("FAILED!!!\n");
    keep_alive(handle,FAIL);
  }
  else
  { printf("PASSED!!!\n");
    keep_alive(handle,PASS);
  }

  clock_ticks = clock();
  printf("Elapsed time = %f seconds.\n",clock_ticks / CLK_TCK);

  if (monitorFlag)
     monitor();

  cleanup();//release DFP control, close files
  return(0);
} //end main


/*******************************************************/
/*          Program Functions                          */
/*******************************************************/

//++++++++++ open_com_port function ++++++++++
//Function:  Open the DFP computor com port and set paremters.
//Pre:   No preconditions
//Post:  DPF computer com port open.
//       port parameters: 57600 baud
//                        no parity
//                        8 bits
//                        1 stop bit
//++++++++++++++++++++++++++++++++++++++++++++++
void
open_com_port (void)
{
  //open com port
  if ((handle = open("com1",(int)(O_BINARY+O_RDWR))) == -1)
    {
    printf("Error opening COM port\n");//not send to 18xx,
                           //port not open!
    exit(PTPROG_PRT_ERROR);
    }
  //set up parameters for com port
  if (!serial_set(handle,BAUD57600,PARITY_NONE,LENGTH_8,STOPBIT_1,
               PROT_NONE,0,0))
    {
    printf("Error setting port parameters\n");//not send to 18xx,
                                  //port parameters not ok!
    close(handle);
    exit(PTPROG_PARAM_ERROR);
    }
  return;
}

//++++++++++ get_args function ++++++++++
//Function:  Get the optional argument string for ptprog.c.
//
//  This functions seperates the "starting switch"
//  statements for how the program was started from the
//  optional arguments for the program.
//
//  Two possibilites for starting ptprog are
//       1. slave.
//       2. DFP keyboard.
//
//  Slave ALWAYS inserts -s into the argument string to
// alert ptprog that slave initiated the program -- and
// therefore the REST OF THE ARGUMENTS are passed
// following the -s (seperated by space/s).
//
//  For debug when you are starting ptprog from the
// DFP keyboard, you may want to sometimes have the
// arguments come from the keyboard, or sometimes
// from the port (18xx testsheet).
//
// The possible "start switch" combinations:
//       1. ptprog started by slave:
//           ptprog -s (args from slave)
//       2. ptprog started from DFP keyboard:
//           ptprog -s (args from keyboard --emulating slave)
//           ptprog -k (agrs from keyboard)
//           ptprog -km(args keyboard)
//           ptprog -p (args from port --18xx testsheet)
//           ptprog -pm(args from port --18xx testsheet)
//
// (switch meanings: s=slave, k=keyboard, p=port, m=monitor)
//
// Examples with arguments:
//       1. start by slave
//               ptprog -s [arg1] [arg2]
//       2. start from DFP keyboard - type:
//               ptprog -s [arg1] [arg2] <enter>
//               ptprog -k [arg1] [arg2] <enter>
//               ptprog -km [ar1g] [arg2] <enter>
//               ptprog -p  <enter>
//                   18xx testsheet <press start>
//                          [arg1] [arg2]
//               ptprog -pm <enter>
//                   18xx testsheet <press start>
//                          [arg1] [arg2]
//
//  This function will fill the variable args[] with only
// the arguments following the switches for where the program
// was started: the -s, -k, -km, -p, -pm are NOT saved.
// Furthermore, the 'P' command and subdirectory name are NOT
// saved from the port string (18xx string).
//       Example: ptprog -km hello world
//            -> args = "hello world"
//       Example: ptprog  -k
//            -> args = ""  (the null string)
//       Example: ptprog  -p
//            18xx sends string:  Pmod1 hello world\n
//            -> args = "hello world"
//
//Pre:   Ptprog is started from the command line or via Slave.
//       Optional arguments are sent from the command line or via
//          the com port.
//
//Post:  Fills the variable args[] with the argument string.
//  Updates global variable monitorFlag.
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
get_args (int argc, char **argv, char *args)
{
  char *arg_ptr;
  int  idx;

  //initialize to default condition
  monitorFlag = 0;

  if (argc < 2 || argv[1][0] != '-')
    usage();
  switch (argv[1][1])
  {
     case 'k': if (argv[1][2] == 'm')
          monitorFlag = 1;
     case 's': *args = 0x0;
          for (idx=2; idx<argc; idx++)
          { if (idx != 2)
            strcat(args," ");
          strcat(args,argv[idx]);
          }
          break;
     case 'p': if (argv[1][2] == 'm')
          monitorFlag = 1;
          puts("\nWaiting for arguments from 18XX.");
          puts("Press ESC to exit.\n");
          if(read_port(handle,args,0))
          exit(0);
          //Remove charactors up to the first space
          arg_ptr = strstr(args," ");
          if (arg_ptr == NULL)
          *args = 0x0;
          else
          strcpy(args,arg_ptr + 1);
          break;
     default:  usage();
  }
  return;
}

//+++++++++++ usage function  +++++++++++++++++
//Function:  Display the switch statements allowed
// for how to start this program from
// the command line.
//Pre:Ptprog.exe is started from the command line
// or by Slave.
//Post: Displays the usage argument switches for starting
// ptprog.exe.
// +++++++++++++++++++++++++++++++++++++++++++++++
void
usage(void)
{
   printf("\n%s%s%s%s%s%s%s%s%s%s%s%s",
   "USAGE: ptprog <switch> [arguments]\n",
   "           switch\n",
   "             -s       Emulate input from Slave via DFP keyboard,\n",
   "                      no Monitor option.\n",
   "             -k[m]    Arguments taken from DFP command line,\n",
   "                      optional Monitor.\n",
   "             -p[m]    Wait for arguments from port (18xx Worksheet),\n",
   "                      optional Monitor.\n",
   "             -h       Help\n",
   "             -?       Help\n",
   "      Note: Switch must be lower case!!!\n");
   exit(0);
}


//++++++++++ open_pt2ini function ++++++++++
//Function:  Opens, reads, closes the pt2.ini file.
//      Calls the read_ini_data function to extract
//       specific information from record/s.
//Pre:none
//Post: If pt2.ini file not exist-> program exits.
// Otherwise:
//      1. fills the local variable irec with the ini.
// 2. Stores specific info from record/s via the
//       read_ini_data function.
//++++++++++++++++++++++++++++++++++++++++++++++
void
open_pt2ini(void)
{
  int error=0;
  //open pt2.ini file
  if ((ini_file = fopen("pt2.ini","rt")) == NULL)
  { sprintf(msg,"Error opening pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }


  // Read all the lines in the pt2.ini file
  while( fgets( irec, I_REC_SIZ - 1, ini_file ) != NULL )
  { if ((error = read_ini_data(irec)) != 0)
    { if (error == WIDTH_ERROR)
        sprintf(msg,"Error reading pt2.ini--WIDTH_ERROR");
      else
        sprintf(msg,"Error reading pt2.ini\n");
      print_Message(msg);
      cleanup();
      exit(error);
    }
  }

  //check really at end of file and not file reading error
  if( !feof( ini_file ) )
  {
    sprintf(msg,"Error reading pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }

  fclose(ini_file);
  return;
}

//++++++++++ read_ini_data function ++++++++++++++
//Function:  Looks for 'L' tag record. The L tag
// record had device specific information.
// For this program the function gets the
// device type and memory size.
//
//Pre:Recieves a record from the pt2.ini file.
//
//Post: If the record is an L record:
//       1. variable device_type =  device type
//       2. variable last_address =  memory size
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++
int
read_ini_data(char *record)
{
  charbytepos[2],
   mem_size[10],
   device_type[30],
   *ptr;

  // Process L type entries
  if (toupper(record[0]) == 'L')
  {
    // Get device type, memory size and byte position
    get_field(3,device_type,record);
    printf("Device programming = %s\n",device_type);
    get_field(6,mem_size,record);
    last_address = atol(mem_size);
    printf("Last Address = %lX Hex\n",last_address);
  }
  return 0;
}

//++++++++++ init_dfp_cards function ++++++++++++++
//Function:  Initialize DFP CCC cards and
//       DR2P cards for the DFP program.
//
//Pre:The appropriate CCC/DR2P cards are installed
// in the DFP and 18xx.
//Post: The CCC + DR2P cards are initialized for
// serial mode to program the 68HC11 via the
// device pins: Reset, ModA, ModB, RxD, and Txd.
//
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++
void
init_dfp_cards(void)
{
  pt2init();   // DFP init
  initCard(CD);// card 0 init
  releaseDR(CD);// This will clear all relays
  setMode(CD,DATA);// card 0 data mode
  ptEnable(CD,ON);// enable DFP, card 0
  outp(DATA_BUS_DIR,0xFD );//Port C pins- 1=output, 0=input
  delay(0); // Calibrate delay

  // Set relays
  outpw(CD_ADDR|GA_INST,D_REED|nodA[16]|SET); // CS
  outpw(CD_ADDR|GA_INST,D_REED|nodA[17]|SET); // SO
  //outpw(CD_ADDR|GA_INST,D_REED|nodA[18]|SET);     // WP
  outpw(CD_ADDR|GA_INST,D_REED|nodA[19]|SET); // SI
  outpw(CD_ADDR|GA_INST,D_REED|nodA[20]|SET); // SK
  //outpw(CD_ADDR|GA_INST,D_REED|nodA[21]|SET);     // HOLD

         //note WP=HOLD=pullup
  return;
}

//++++++++++ store_data function ++++++++++
//Function: Store the 'write' data to the
//       writedat[] array.
//       1. load the writedat[] array with FF's
//       2. load the serial number in positions
//             0-7
//       3. load the date in positions 8-10
//
//Pre:The program arguments are stored in args variable
//Post: The writedat[] array is loaded.
//+++++++++++++++++++++++++++++++++++++++++
static int
store_data (void)
{
  struct dosdate_t ddate;
  int len=0;
  int i;
  char**parg;  //array of pointers

  // ** break up the argument string into individual strings **
  parg=breakUp(args);  //refer to pt2.h breakUp

  // ** init the writedat array **
  for (i=0; i<last_address; i++)
    writedat[i]=(unsigned char) 0xFF;

  // **  get the serial number **
  //serial number not 8 digits?
  len=strlen(parg[0]);
  if (len != 8)
    return(ERROR);

  //write serial number to writedat array
  // and check for alpha-numeric input
  for (i=0; i<8; i++)
  {
    if (isalnum(parg[0][i]))
      writedat[i]=(unsigned char) parg[0][i];
    else
      return(ERROR);
  }

  // **  get the date **
  _dos_getdate(&ddate);
  printf("\ndate= %2.2x %2.2x %2.2x\n",
               ddate.day,ddate.month,ddate.year-1900);
  i=8;
  writedat[i]=(unsigned char) ddate.day;
  i++;
  writedat[i]=(unsigned char) ddate.month;
  i++;
  writedat[i]=(unsigned char) (ddate.year-1900);

return(NOERROR);
}

//++++++++++ program_memory function ++++++++++
//Function: Program eeprom memory
//       1. unprotect all addresses-able to write to eeprom
//       2. do either a page write or
//             byte write depending on the type
//             variable.
//       3. protect the eeprom from future writes
//
//Pre:The program arguments are pointed to by parg[].
// (this program only expects one argument-> the serial number)
//Post: The writedat[] array is loaded.
//+++++++++++++++++++++++++++++++++++++++++
int
program_memory (int type)
{
  int n;
  wrsr();  //unprotect all addresses via status register

  printf("programming eeprom\n");

  //if page write
  if (type == PAGE_WRITE)
  {
    for (n=0; n<last_address; n=n+4)
    {
      wren();
      if (pagewr(n))
        return ERROR ;
    }
  }

  //if byte write
  if (type == BYTE_WRITE)
  {
    for (n=0; n<last_address; n++)
    {
      wren();
      if (bytewr(n, writedat[n]))
        return ERROR;
    }
  }

  wrdi();//write disable

  return NOERROR;
}

//++++++++++ verify_memory function ++++++++++
//Function: verify programming of eeprom.
// 1. read eeprom contents and store into readdat[] array
// 2. compare readdat[] array with writdat[] array
//
//Pre:None
//Post: Returns:
//       ERROR   - data not match
//       NOERROR - data match
//+++++++++++++++++++++++++++++++++++++++++
int
verify_memory (void)
{
  int i;

  printf("verifing eeprom\n");
  //read entire eeprom and store into readdat[] array
  rd(0,last_address);

  //compare readdate[] array and writedat[] array contents
  for (i=0; i<last_address; i++)
  {
    if (readdat[i]!=writedat[i])
    {
      printf("\nAddr error = %x, wdata = %x, rdat = %x\n",
                          i,writedat[i],readdat[i]);
      return ERROR;
    }
  }

  return NOERROR;
}

//++++++++++ write enable function ++++++++++
//Function: Writes the eeprom enable instruction.
//
//Pre:None
//Post: programing eeprom is enabled
//+++++++++++++++++++++++++++++++++++++++++
void
wren (void)
{
  int i;

  //instruction format=0000 0110
  for(i=0; i<5; i++)
    si_low();
  si_high();
  si_high();
  si_low();

  disable_chip();
}

//++++++++++ write disable function ++++++++++
//Function: Writes the eeprom disable instruction.
//
//Pre:None
//Post: programing eeprom is disabled
//+++++++++++++++++++++++++++++++++++++++++
void
wrdi (void)
```

```
{
  int i;

  //instruction format=0000 0100
  for(i=0; i<5; i++)
    si_low();
  si_high();
  si_low();
  si_low();

  disable_chip();
}

//++++++++++ read status reg function ++++++++++
//Function: Reads the eeprom status registar
//       and returns the status
//
//Pre:None
//Post: Returns: unsigned char- status
//++++++++++++++++++++++++++++++++++++++++++
unsigned char
rdsr (void)
{
  int i, data1;
  unsigned charstatus;

  //instruction format=0000 0101
  for(i=0; i<5; i++)
    si_low();
  si_high();
  si_low();
  si_high();

  //read and store status data
  data1=read_8_bits();
  status= (unsigned char) data1;

  disable_chip();
  return status;
}

//++++++++++ write status reg function ++++++++++
//Function: Writes to the eeprom status registar
//       to unprotect all addresses.
//
//Pre:None
//Post: All eeprom addresses are unprotected.
//++++++++++++++++++++++++++++++++++++++++++
void
wrsr (void)
{
  int i;

  //instruction format=0000 0001
  for(i=0; i<7; i++)
    si_low();
  si_high();

  //BPI=0, BPO=0 - no addresses protected
  for (i=0; i<7; i++)
    si_low();

  disable_chip();
}

//++++++++++ read function ++++++++++
//Function: Reads data from the eeprom.
// The function recieves the
// starting and ending address to
// read, and fills the readdat[] array.
//
//Pre:Receives the starting and ending addresses.
//Post: The data is written to the readdat[] array.
//+++++++++++++++++++++++++++++++++++++++++
void
rd (int start_addr, int end_addr)
{
  int i;
  unsigned int data1;

  //instruction format=0000 A011
  for(i=0; i<4; i++)
    si_low();
  if(start_addr > 255)//is A8=1 or =0?
    si_high();
  else
    si_low();
  si_low();
  si_high();
  si_high();

  write_8_bits(start_addr);//write address to start read from

  //read and store eeprom data from starting address to ending address
  for (i=start_addr; i<end_addr; i++)
  {
    data1=read_8_bits();
    readdat[i]= (unsigned char) (data1 &  0xFF); //cast as unsigned char
  }

  disable_chip();
}

//++++++++++ byte write function ++++++++++
//Function: Write 1 bytes to the eeprom
//       via the byte write instruction.
//       The function receives the
//       address.
```

```
//
//Pre:Receives the starting address
//Post: the byte is written to the eeprom
//+++++++++++++++++++++++++++++++++++++++++
int
bytewr (int addr, unsigned char wrdat)
{
  int i;
  unsigned char status;

  //instruction format=0000 A010
  for(i=0; i<4; i++)
    si_low();
  if(addr > 255)//is A8=1 or =0?
    si_high();
  else
    si_low();
  si_low();
  si_high();
  si_low();

  write_8_bits(addr);//write address to eeprom
  write_8_bits((int) wrdat);//write data to eeprom

  disable_chip();

  for (i=0; i<10; i++)
  {
    delay (1);
    status=rdsr(); //read status reg
    if((status & 0x0001)==0x0000) //check status bit 0
      return(NOERROR);
  }

  return (ERROR);
}

//++++++++++ page write function ++++++++++
//Function: Write 4 bytes to the eeprom
//       via the page write instruction.
//       The function receives the
//       starting address and each byte
//       is written to the next consequetive
//       address for 4 addresses.
//
//Pre:Receives the starting address
//Post: 4 bytes are written to the eeprom
//+++++++++++++++++++++++++++++++++++++++++
int
pagewr (int addr)
{
  int i;
  unsigned char wrdat;
  unsigned char status;
  //instruction format=0000 A010
  for(i=0; i<4; i++)
    si_low();
  if(addr > 255)//is A8=1 or =0?
    si_high();
   else
    si_low();
  si_low();
  si_high();
  si_low();

  write_8_bits(addr);//write address to eeprom

  //write data to eeprom
  for(i=addr; i<addr+4; i++)
  {
    wrdat=writedat[i];
    write_8_bits((int) wrdat);//write data to eeprom
  }

  disable_chip();

  for (i=0; i<10; i++)
  {
    delay (1);
    status = rdsr(); //read status reg
    if((status & 0x0001)==0x0000) //check status bit 0
      return (NOERROR);
  }
  return (ERROR);

}

//++++++++++ read_8_bits function ++++++++++
//Function:  reads 8 bits of data from the
// eeprom, MSB to LSB, 1 bit at a time.
//
// 1. read a bit (output SO)
// 2. shift the output (SO) to the bit 0 position (in this case
//       shift right 1 bit).
// 3. mask it with 0x01.
// 4. shift it to the appropriate position (shift left 'i' times).
// 5. add it to the previous result in data1 variable.
//
// ex:   read first bit (MSB) with data1 initialized to 0x00
//
//       read (SO=bit1)       : 0x02                 (0000 0010)
//       shift right 1 (>>1)    : 0x02 -> 0x01         (0000 0001)
//       mask with 0x01         : 0x01 '&' 0x01 = 0x01 (0000 0001)
//       shift to 7 to left     : 0x01 -> 0x80         (1000 0000)
//       add to data1           : 0x00 '+' 0x80 = 0x80 (1000 0000)
//PRE:   NONE
//POST:  return: int - contains data read
```

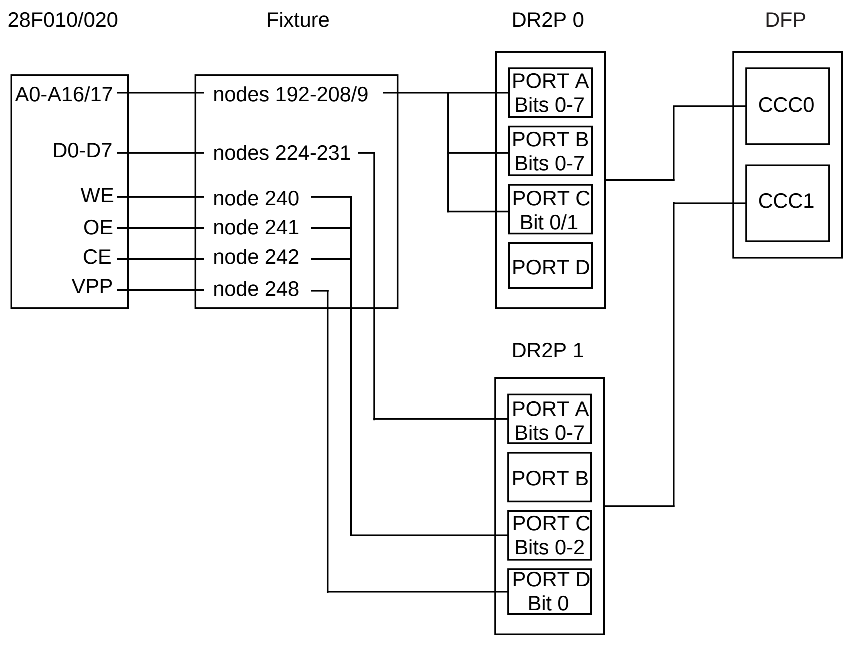
```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++
int
read_8_bits()
{
  int i, data1=0;
  for (i=7; i>=0; i--)
  {
    si_low();
    data1 = data1 + (((inpw(DATA_BUS) >> 1) & 0x01) << i);
  }
  return data1;
}

//++++++++++ write_8_bits function ++++++++++
//Function: write 8 bits to eeprom, 1 bit at a time,
//  MSB->LSB.
//  1. shift each data bit to the bit 0 position.
//  2. mask it with 0x01.
//  3. check if it is true (high) or false (low).
//
//PRE:   receives the 8 bits of data to be written
//POST:  data written 1 bit at a time to eeprom
//+++++++++++++++++++++++++++++++++++++++++++
void
write_8_bits (int data)
{
  int m;
  for (m=7; m>=0; m--)
  {
    if((data >> m) & 0x01)
      si_high();
    else
      si_low();
  }
}

//++++++++++ disable chip function ++++++++++
//Function: chip select is high for
//       one clock (SK low to SK high)
//
//note: bit7, bit6, bit5,  bit4, bit3, bit2, bit1, bit0
//      X     X     HOLD*  SCK   SI    WP*   SO    CS*
// each bit level (high or low) is defined in the header.
//
//Pre:CCC and DR2P cards have been set up.
//Post:CS = high
//+++++++++++++++++++++++++++++++++++++++++++
void
disable_chip (void)
{ outp(DATA_BUS, HD_H | SK_L | SI_L | WP_H | SO_L | CS_H);  //clock low
  outp(DATA_BUS, HD_H | SK_H | SI_L | WP_H | SO_L | CS_H);  //clock high
}

//++++++++++ si_low function ++++++++++
//Function: Input a LOW on SI:
//       SI is held low for
//       one clock (SK low to SK high)
//       while CS is low.
//
//note: bit7, bit6, bit5,  bit4, bit3, bit2, bit1, bit0
//      X     X     HOLD*  SCK   SI    WP*   SO    CS*
// each bit level (high or low) is defined in the header.
//
//Pre:CCC and DR2P cards have been set up.
//Post:CS = high
//++++++++++++++++++++++++++++++++++++++++++
void
si_low (void)
{ outp(DATA_BUS, HD_H | SK_L | SI_L | WP_H | SO_L | CS_L);
  outp(DATA_BUS, HD_H | SK_H | SI_L | WP_H | SO_L | CS_L);
}

//++++++++++ si_high function ++++++++++
//Function: Input a HIGH on SI:
//       SI is held high for
//       one clock (SK low to SK high)
//       while CS is low.
//
//note: bit7, bit6, bit5,  bit4, bit3, bit2, bit1, bit0
//      X     X     HOLD*  SCK   SI    WP*   SO    CS*
// each bit level (high or low) is defined in the header.
//
//Pre:CCC and DR2P cards have been set up.
//Post:CS = high
//++++++++++++++++++++++++++++++++++++++++++
void
si_high (void)
{ outp(DATA_BUS, HD_H | SK_L | SI_H | WP_H | SO_L | CS_L);
  outp(DATA_BUS, HD_H | SK_H | SI_H | WP_H | SO_L | CS_L);
}


//+++++++++++ monitor function  +++++++++++++++++
//Function:  Displays the contents of DUT memory, 16 addresses
//       at a time, on the DFP terminal.
// The user enters enters an address 0 (zero) to
//       highest DUT address via the DFP keyboard.
//      The address are in hex -- example:  1c
// The contents of 16 addresses of DUT memory will be displayed,
//       starting with the entered address.
//       example -- user enters:   30
//             the contents of address 30 hex (0x30)
//             is the first data displayed, followed by
//             the contents of the next 16 addresses.
//
// This is an extremely simple monitor.
// Error tolerance is high, and error handling is therefore nil.
// User cannot backspace. If he errs in typing the address -
//       just hit return, or the space bar, and try again.
// Entering an 'X' (exit) or 'Q' (quit) will terminate
//       the function.
// As long as no 'X' or 'Q' are recieved, the user may
//       continue to enter addresses to display the
//       Dut memory contents.
//
//Pre:Ptprog.exe is started from the command line
// with the starting argument -km or -pm.
// (see get_args function.)
//Post: The data contents of 16 DUT address locations are
// displayed, starting with the user input address.
// +++++++++++++++++++++++++++++++++++++++++++++++++++++
void
monitor(void)
{
  unsigned char t;// misc counting variable
  unsigned char byte; // location to store data
  unsigned long adr; // the "address" part of the query to DUT
  int count; // counts the bytes returned from DUT
  char cmd[80];// keyboard input string
  char done = 0;// a loop control item
  char *endptr;// needed for strtoul function (string to
                   //unsigned long)

  while (!done)
  {
    printf("\n\aDUT Monitor.  Enter X to quit.\n");
    printf("->"); // prompt:
    scanf("%s",cmd);

    // Convert input string to upper case.
    for (count = 0; count < 6 ; count++)
    {
      cmd[count] = (char)toupper(cmd[count]);
    }

    // These are the exit criteria: x, X, q, or Q.
    if ( (cmd[0] == 'X') || (cmd[0] == 'Q'))
    {
      done = 1;
      printf("\aExiting monitor\n");
      break;
    }

    //convert ascii string into unsigned long (hex= base 16)
    adr = strtoul(cmd, &endptr, 16);
    printf("adr: %05.5lX\n",adr); // verify address.

    // read 16 bytes
    rd(adr,adr+16);

    for (t=0; t<16; t++)
      printf("%2.2x ",readdat[t]);
  } // end of while !done
} // end of Monitor

//++++++++++ cleanup function ++++++++++++++
//Function:  Release the DR2P cards from
// DFP control, close all files+ handles.
//
//Pre:Program is about to exit (return).
//Post: DFP releases DR2P control, all
// files + handles closed.
//+++++++++++++++++++++++++++++++++++++++++++++++++
void
cleanup(void)
{
  releaseDR(CD);// Disable PrompTest
  close(handle);// Close com port handle
  fcloseall();// Close any open files
  return;
}

//++++++++++ process message function ++++++++++++++
//Function:  print failure/error message
//       to both DFP and 18xx screen
//
//Pre:Receives msg buffer with formatted message string.
//Post: Prints message to DFP screen.
// Sends message to 18xx to printed to 18xx screen.
//+++++++++++++++++++++++++++++++++++++++++++++++++
void
print_Message(char *msg)
{
   printf("%s",msg); //print to DFP screen
   send18xxMsg(handle,DFP_18XX_MSG,msg); //send to 18xx DigFuncProc
                                 //worksheet
  return;
}
```

# Custom Example — Parallel Flash in Free Air 6

The 18XX DigFuncProc testsheet sends the subdirectory path to SLAVE.EXE on the DFP computer and to start the appropriate PTPROG.EXE (there are no arguments for this application). PTPROG.EXE determines which flash memory it is programing (28F010 or 28F020) via the PT2.INI file. PTPROG.EXE erases the flash if necessary, then writes the customer data (the .img file) to the flash. The program then verifies the write and then sends the PASS/FAIL information to the 18XX DigFuncProc testsheet.

Ptprog.exe can also be started from the DFP keyboard.

Figure 6.1 Parallel Flash In Free Air Interconnect Diagram



## PT2.INI File

Below is the PT2.INI file used in the 28F010 example.

```
L,IC1,28F010,test.dat,87,131072,0,1
R,note-format 87 = Motorola S record type
M,89,B4
```

Below is the PT2.INI file used in the 28F020 example.

```
L,IC1,28F020,test.dat,87,262144,0,0
R,note-format 87 = Motorola S record type
M,01,2A
```

## PTPROG.C File

Below is the ptprog.c file used in this example.

```
/*
*** Parallel Flash Example ***

Custom Application For programming 28F010/28F020 Flash Memory
Filename: ptprog.c
Component/s: 28F010, 28F020
Aliases        : AM28F010, i28F010, AM28F020, i28F020

Device Manufacturer: AMD, Intel
Device Function: 28F010:131072 x 8 Flash Memory
               : 28F020:262144 x 8 Flash Memory
IC Package: 32 Pin DIP
Written by: Sidney Fluhrer - Teradyne - ATWC
              : Barbara Ryan - Teradyne - ATWC
Date Created: 12-OCT-95
Original Source:
Fixture Requirements: see Wiring below
Files Required: ptprog.exe (compiled ptprog.c)
                 pt2.ini    (see pt2.ini below)
                 data file  (see pt2.ini below)
Written with OS: DFP  - B.0
                 18xx - F.0
                 To use this program with earlier versions
                     (either DFP or 18xx)
                     comment out the 'send18xxMsg()'
                     function call in the
                     print_Message() function.
```

```
Test Description:

   28F010 Flash Memory:1M (131072 x 8-Bit)
   28F020 Flash Memory:2M (262144 x 8-Bit)
                     CMOS
                     12.0 Volt-write/erase cycles
                     Bulk Erase

   This program can be used for either the 28F010 or
   28F020 since only the memory size is the differing
   factor.  The program determines which device it is
   currently testing via the memory size and device
   type in the pt2.ini file. (See pt2.ini examples below.)

   The program will check for an erased state, erase if
   necessary, program per customer provided data, and
   verify the programming.

Program Organization Outline:
   1. open com ports/set up parameters
   2. get program arguments
   3. check arguments
         note: 1 optional program argument allowed:
               arg = e  or  v
                  where:
                     e = erase device only
                     v = verify device only
   4. open, read, close pt2.ini file - retrieve:
                     device type
                     device size (last_address)
                     byte position
                     mfg id and device code
                     data .img file name
   5. open data file
   6. set up CCC/DR2P cards
   7. set up timers
   8. initilize device
   9. check device for valid mfg id/device code
   10. check device for erased condition
         note- if NOT erased - check if
               programed data is VALID (verify)
   11. erase device if necessary
   12. program device
   13. verify device -
               verify via reading each address and
                     checking the byte against the
                     corresponding data .img file byte
   14. cleanup:
         write PASS/FAIL to 18xx (pcom.exe)
         enter monitor mode if monitor flag true (only if
               program started from DFP keyboard for
               debug)
         close all file and ports
         release DR's
         exit(0)

18xx Worksheet/Argument usage:
   1. DigFuncProc worksheet example:
               Source Dir:mod1
               Arguments:
               Timeout:5

         mod1    = subdirectory to find ptprog.exe
                 = no arguments = default (see below)
         9     = timeout in seconds for DigFuncProc worksheet

   Arguments:
   1. no arguments ->default = erase/program/verify.
   2. e        ->erase only (will not program or verify)
   3. v        ->verify only (will not erase or program)

Sample pt2.ini - 28F010:
   L,IC1,28F010,ic1.dat,87,131072,0,1
   M,89,B4

   where:L     = local device tag
         IC1   = board identifier
         28F010= device type
         ic1.dat = data source file
         87    = format of data source file (87 = motorola s-record)
         131072= memory size
         0     = byte position
         1     = fill character (fill s-record data with FF)

         M     = manufacturer tag
         89    = manufacturer code
         B4    = device code

Sample pt2.ini - 28F020:
   L,IC2,28F020,ic2.dat,87,262144,0,1
   M,01,2A
```

```
Wiring          :

   signal:     DFP card:DFP node/s:
   -------     --------------------
   A0-A16/17card 0   192-208/209 (Port A, B, C)
   D0-D8       card 1      224-231     (Port A)
   WE*         card 1      240         (Port C)
   OE*         card 1      241         (Port C)
   CE*         card 1      242         (Port C)
   VPP         card 1      248         (Port D)

Statistics-28F010/020:
   18xx:       O.S. Version - E.3/F.0
   DFP:        O.S. Version - A.3/B.0
               AmiBios - ISA BUS CLOCK SELECTION = AUTO
               AmiBios - DRAM = FAST
               SmartDrv - Set (times taken after 1 initial
                      run to load smart drv memory)
   Compiler:Turbo C - 3.0
   sample of: 5P28F010 -120 (Intel)
   sample of: 2AM28F020 -150 (AMD)
   aprox. times for programming an erased 28F010/020:
         28F010 = 131072 x 8 bits
         28F020 = 266144 x 8 bits

                   28F010      28F020
   setup             .110        .110s
   device/mfg id check  .110   .110s
   check all erase  .600    1.100s *
   program            4.500       8.850s
   verify all   1.000       1.900s
   cleanup            .110        .110s
   18xx/DFP memory swap  .200   .200s
                   -----        -----
   Total program 6.630     12.380sec

   (*NOTE: a bulk erase for 28F010 may add aprox. 3-11 sec)
           a bulk erase for 28F020 may add aprox. 8-20 sec)

*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <fcntl.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <dos.h>
#include <errno.h>
#include <ctype.h>

#include "pt2.h"
```

```
#define MAX_WIDTH1// One device and datafile to
                     // be identified via the pt2.ini file

//CCC+DR2P control
//note:  bus or channel direction (input/output) is from
//  the DFP's  point of view
#define CTC_CARDCARD1//CTC timer CCC card 1
#define INC_ADDRCARD0|ADD_INC//increment address bus card 0
#define CNTL_BUSCARD1|PC_DATA//control bus data = C bus, card 1
#define DATA_BUSCARD1|PA_DATA//data bus data = A bus, card 1
#define DATA_DIRCARD1|PAB_CNTL//data bus control (A,B direction)
#define OUTPUT0xFE30 //data bus output FROM DFP,
                             // + VPP relay closed
#define INPUT0xFE33  //data bus input TO DFP,
                             // + VPP relay closed

//device control - read, write, chip enable
#define WE_L0x00     //WE* = low
#define WE_H0x01     //WE* = high
#define OE_L0x00     //OE* = low
#define OE_H0x02     //OE* = high
#define CE_L0x00     //CE* = low
#define CE_H0x04     //CE* = high


static charnotice[]={"Digital Function Processor system"
                  " Copyright (C) 1993,1994 Teradyne Inc.\n"},
         filename[2][13],//store up to 2 data file names
         msg[I_REC_SIZ];//message buffer

static FILE*ini_file,//point to pt2.ini file
         *failfile,  //point to fail file
         *datafile[2];//able to point to 2 data files

static inthandle,    //com port handle
         code_count, //number of mfg/id codes allowed
         error,
         monitorFlag,
         eraseFlag,
         programFlag,
         verifyFlag;

static longlast_address,
         program_count;//total program memory writes

static unsigned charmfg_code[I_REC_SIZ],
               device_code[I_REC_SIZ];

static unsigned intfailval;

/* Prototypes */

//* general functions
```

```
int main(int argc, char **argv);
static void open_com_port(void);
static void get_args (int argc, char **argv, char *args);
static void usage(void);
static void check_args(char *args);
static void open_pt2ini(void);
static int read_ini_data(char *record);
static void open_data_file(void);
static void read_datafile(FILE *datafile, unsigned char *data_byte);
static void init_dfp_cards(void);
static void ccc_timer_setup(void);
static void delay_7us (void);
static void delay_10ms (void);
static void monitor(void);
static void cleanup(void);
static void print_Message(char *msg);//print msg to DFP + 18xx screen

//* functions for this device
static void reset_command(void);
static void turn_on_VPP(void);
static void check_product_code(void);
static int check_erase (long start_addr, long end_addr);
static void erase_memory (void);
static void program_memory (long start_addr, long end_addr);
static void program_byte (unsigned char data);
static int verify_memory(void);
static void write_byte(unsigned char data);
static unsigned char read_byte(void);
static void enable_read_byte(void);
static void disable_read_byte(void);
static unsigned char read_byte_only(void);

/*******************************************************/
int
main(int argc, char **argv)
{
  char   args[MAX_COM_SIZ];
  clock_tclock_ticks;

  failval=0;// initialize failure flag

/*******************************************************/
/* Open DFP COM port + set port parameters             */
/*******************************************************/

  open_com_port();

/*******************************************************/
/* Display copyright notice                            */
/*******************************************************/

  printf("\n%s",notice);

/*******************************************************/
```

```
/* Create argument string + check/store arguments       */
/*******************************************************/
  get_args (argc, argv, args);
  printf("The argument string = '%s'.\n",args);

  check_args(args);//sets up program variables:
                   //eraseFlag
                   //programFlag
                   //verifyFlag

/*******************************************************/
/* Open, read, close pt2.ini file                      */
/*******************************************************/

  open_pt2ini();

/*******************************************************/
/* Open Data File/s                        */
/*******************************************************/

 open_data_file();

/*******************************************************/
/* Initialize Channel Control Cards + DR2P Cards       */
/*******************************************************/

  init_dfp_cards();//initialize CCC + DR2P cards

  // Setup 7us and 10ms delay timers
  ccc_timer_setup();

/*******************************************************/
/* Initialize Device                                   */
/*******************************************************/

  // Initialize device -> Enable Device
  outp(CNTL_BUS, CE_H | OE_H | WE_H);
  outp(CNTL_BUS, CE_L | OE_H | WE_H);
  reset_command();//start out with device in known condition

/*******************************************************/
/* Read Product Identification                         */
/*******************************************************/

  check_product_code();

/*******************************************************/
/* Check Erased Condition                              */
/*******************************************************/

  error = NOERROR;//init error variable
  if(!failval && eraseFlag)
  {
    printf("Checking if Flash is Erased.\n");
    error = check_erase (0x0000, last_address);
    if (error==NOERROR)
      eraseFlag=0;
  }

/*********************************************************/
/* If NOT erased - check if contents are good           */
/*********************************************************/

  if (!failval && error && verifyFlag)
  {
    printf("Flash not Erased -> Verifying Flash.\n");
    error = verify_memory();//datafile verification
    if (!error)//if data verified-program is done!
    {
      printf("Memory Verified!!\n");
      eraseFlag=0;
      programFlag=0;
      verifyFlag=0;
    }
  }

/*********************************************************/
/*  Erase flash                                          */
/*********************************************************/

  if (!failval && eraseFlag)
  {
    printf("Erasing Flash.\n");
    erase_memory();

    if (failval)
    {
      sprintf(msg,"Flash failed to Erase!!\n");
      print_Message(msg);
    }
  }

/*********************************************************/
/* Program Flash Memory                                  */
/*********************************************************/

  if(!failval && programFlag)// Program Flash
  {
    printf("Programming Flash.\n");
    program_memory(0x00000, last_address);

    if (failval)
    {
      sprintf(msg,"Flash failed to Program!!\n");
      print_Message(msg);
    }
  }
```

```
/*******************************************************/
/* Verify Flash Memory                                  */
/*******************************************************/

  reset_command();//Port A out-> Reset Command
  selVpp (1,OFF);// Disable Vpp

  if (!failval && verifyFlag)// Verify Flash
  {
    printf("Verifying Flash.\n");
    error = verify_memory();//datafile verification

    if (error)
    {
      failval=1;
      sprintf(msg,"Flash failed to Verify!!\n");
      print_Message(msg);
    }
  }

  // Disable Device
  outp(CNTL_BUS, CE_H | OE_H | WE_H);

/*******************************************************/
/* Cleanup:                                             */
/* send 18xx PASS/FAIL     */
/* enter monitor mode if flag is set */
/* cleanup() function: release (CLR) DR2p's */
/*                 close com handle */
/*                 close any open files */
/*******************************************************/

  if (failval)
  {
    printf("FAILED!!!\n");
    keep_alive(handle,FAIL);
  }
  else
  {
    printf("PASSED!!!\n");
    keep_alive(handle,PASS);
  }

#ifdef TURBO//if use TURBO C compiler
  clock_ticks = clock();
  printf("Elapsed time = %f seconds.\n",clock_ticks / CLK_TCK);
#endif

  if (monitorFlag)
     monitor();

  cleanup();//release DFP control, close files

  return(0);
```

```
} //end main

/********************************************************/
/*              PROGRAM     FUNCTIONS                   */
/********************************************************/

//++++++++++ open_com_port function ++++++++++
//Function:  Open the DFP computor com port and set paremters.
//   If the port can't be opened or the parameters can't
// be set, the program exits.
//   No message is sent to 18xx, since the com port isn't
// open to send it!
//
//Pre:   No preconditions
//Post:  DPF computer com port open.
//       port parameters: 57600 baud
//                        no parity
//                        8 bits
//                        1 stop bit
//++++++++++++++++++++++++++++++++++++++++++++
void
open_com_port(void)
{
  //open com port 1
  if ((handle = open("com1",(int)(O_BINARY+O_RDWR))) == -1)
  {
    printf("Error opening COM port\n"); //not send to 18xx,
                           //port not open!
    exit(PTPROG_PRT_ERROR);
  }

  //set com port 1 parameters
  if (!serial_set(handle,BAUD57600,PARITY_NONE,LENGTH_8,STOPBIT_1,
               PROT_NONE,0,0))
  {
    printf("Error setting port parameters\n");  //not send to 18xx,
                                 //port parameters not ok!
    close(handle);// Close com port handle
    exit(PTPROG_PARAM_ERROR);
  }

  return;
}

//++++++++++ get_args function ++++++++++
//Function:  Get the optional argument string for ptprog.c.
//
//  This functions seperates the "starting switch"
//  statements for how the program was started from the
//  optional arguments for the program.
//
//  Two possibilites for starting ptprog are
//       1. slave.
//       2. DFP keyboard.
```

```
//
//  Slave ALWAYS inserts -s into the argument string to
// alert ptprog that slave initiated the program -- and
// therefore the REST OF THE ARGUMENTS are passed
// following the -s (seperated by space/s).
//
//  For debug when you are starting ptprog from the
// DFP keyboard, you may want to sometimes have the
// arguments come from the keyboard, or sometimes
// from the port (18xx testsheet).
//
// The possible "start switch" combinations:
//       1. ptprog started by slave:
//           ptprog -s (args from slave)
//       2. ptprog started from DFP keyboard:
//           ptprog -s (args from keyboard --emulating slave)
//           ptprog -k (agrs from keyboard)
//           ptprog -km(args keyboard)
//           ptprog -p (args from port --18xx testsheet)
//           ptprog -pm(args from port --18xx testsheet)
//
// (switch meanings: s=slave, k=keyboard, p=port, m=monitor)
//
// Examples with arguments:
//       1. start by slave
//               ptprog -s [arg1] [arg2]
//       2. start from DFP keyboard - type:
//               ptprog -s [arg1] [arg2] <enter>
//               ptprog -k [arg1] [arg2] <enter>
//               ptprog -km [ar1g] [arg2] <enter>
//               ptprog -p  <enter>
//                   18xx testsheet <press start>
//                          [arg1] [arg2]
//               ptprog -pm <enter>
//                   18xx testsheet <press start>
//                          [arg1] [arg2]
//
//  This function will fill the variable args[] with only
// the arguments following the switches for where the program
// was started: the -s, -k, -km, -p, -pm are NOT saved.
// Furthermore, the 'P' command and subdirectory name are NOT
// saved from the port string (18xx string).
//        Example: ptprog -km hello world
//             -> args = "hello world"
//        Example: ptprog  -k
//             -> args = ""  (the null string)
//        Example: ptprog  -p
//             18xx sends string:  Pmod1 hello world\n
//             -> args = "hello world"
//
//Pre:   Ptprog is started from the command line or via Slave.
//       Optional arguments are sent from the command line or via
//          the com port.
//
```

```
//Post:  Fills the variable args[] with the argument string.
//  Updates global variable monitorFlag.
// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
get_args (int argc, char **argv, char *args)
{
  char *arg_ptr;
  int  idx;

  //initialize to default condition
  monitorFlag = 0;

  if (argc < 2 || argv[1][0] != '-')
    usage();
  switch (argv[1][1])
  {
     case 'k': if (argv[1][2] == 'm')
          monitorFlag = 1;
     case 's': *args = 0x0;
          for (idx=2; idx<argc; idx++)
          { if (idx != 2)
            strcat(args," ");
          strcat(args,argv[idx]);
          }
          break;
     case 'p': if (argv[1][2] == 'm')
          monitorFlag = 1;
          puts("\nWaiting for arguments from 18XX.");
          puts("Press ESC to exit.\n");
          if(read_port(handle,args,0))
          exit(0);
          //Remove charactors up to the first space
          arg_ptr = strstr(args," ");
          if (arg_ptr == NULL)
          *args = 0x0;
          else
          strcpy(args,arg_ptr + 1);
          break;
     default:  usage();
  }
  return;
}

//+++++++++++ usage function  +++++++++++++++++
//Function:  Display the switch statements allowed
// for how to start this program from
// the command line.
//Pre:Ptprog.exe is started from the command line
// or by Slave.
//Post: Displays the usage argument switches for starting
// ptprog.exe.
// ++++++++++++++++++++++++++++++++++++++++++++++
void
usage(void)
```

```
{
   printf("\n%s%s%s%s%s%s%s%s%s%s%s%s",
   "USAGE: ptprog <switch> [arguments]\n",
   "           switch\n",
   "             -s       Emulate input from Slave via DFP keyboard,\n",
   "                      no Monitor option.\n",
   "             -k[m]    Arguments taken from DFP command line,\n",
   "                      optional Monitor.\n",
   "             -p[m]    Wait for arguments from port (18xx Worksheet),\n",
   "                      optional Monitor.\n",
   "             -h       Help\n",
   "             -?       Help\n",
   "      Note: Switch must be lower case!!!\n");
   exit(0);
}

//++++++++++ check_args function ++++++++++
//Function:  Check program arguments and
//       fill variables accordingly.
//     This application accepts 3 possibilities:
// 1.  no arguments - default -erase, program, verify
// 2.  e argument  -  will erase only
// 3.  v argument  -  will verify only
//Pre:   get_args function has screened the program
//       arguments and put them into the
//       string variable args.
//Post:  updates variables:
//                  default   erase   verify
//       eraseFlag1  1     0
//       programFlag10     0
//       verifyFlag1 0     1
//++++++++++++++++++++++++++++++++++++++++++++++
void
check_args(char *args)
{
  //initialize to default condition
  eraseFlag=1;
  programFlag=1;
  verifyFlag=1;

  if (strlen(args)>0)
  {
    if (args[0]=='e')
    {  programFlag=0;
       verifyFlag=0;
       printf("** Erase only **\n");
    }
    else if (args[0]=='v')
    {  programFlag=0;
       eraseFlag=0;
       printf("** Verify only **\n");
    }
  }
  return;
}


//++++++++++ open_pt2.ini function ++++++++++
//Function:  Opens, reads, closes the pt2.ini file.
//      Calls the read_ini_data function to extract
//       specific information from record/s.
//Pre:none
//Post: If pt2.ini file not exist-> program exits.
// Otherwise:
//      1. fills the local variable irec with the ini.
// 2. Stores specific info from record/s via the
//       read_ini_data function.
//++++++++++++++++++++++++++++++++++++++++++++++
void
open_pt2ini(void)
{
  charirec[I_REC_SIZ];//pt2.ini input buffer

  //open pt2.ini file
  if ((ini_file = fopen("pt2.ini","rt")) == NULL)
  { sprintf(msg,"Error opening pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }


  // Read all the lines in the pt2.ini file
  while( fgets( irec, I_REC_SIZ - 1, ini_file ) != NULL )
  { if ((error = read_ini_data(irec)) != 0)
    { if (error == WIDTH_ERROR)
        sprintf(msg,"Error reading pt2.ini--WIDTH_ERROR");
      else
        sprintf(msg,"Error reading pt2.ini\n");
      print_Message(msg);
      cleanup();
      exit(error);
    }
  }

  if( !feof( ini_file ) )
  { sprintf(msg,"Error reading pt2.ini\n");
    print_Message(msg);
    cleanup();
    exit(INI_ERROR);
  }
  fclose(ini_file);
  return;
}


//++++++++++ read_ini_data function ++++++++++++++
//Function:  A. Looks for 'L' tag record. The L tag
```

```
// record had device specific information.
// For this program the function gets the
// device type, memory size, device position, and
// data filename/s. The data filename is processed
// to drop any name extensions and adds the .img
// extension.  Also the device type and memory
// sized are checked for a "match" since this
// program can be used for 28F010 or a 28F020.
//      B. Looks for 'M' tag record.  The M
// tag record has mfg id and device code
// information.
//
//Pre:Recieves a record from the pt2.ini file.
//
//Post: If the record is an L record:
//       1. variable device_type =  device type
//       2. variable last_address =  memory size
//       3. variable bytepos =  byte position
//       4. variable filename[][] = datafile/s with .img extension
// If record is M record:
//       1. variable mfg_ptr = mfg codes
//       2. variable id    = device codes
//       3. variable code_count = number of mfg/device code pairs
//
//++++++++++++++++++++++++++++++++++++++++++++++++++++
int
read_ini_data(char *record)
{
  charimgfile[13],
   bytepos[2],
   mem_size[10],
   code[I_REC_SIZ],
   device_type[30],
   *ptr;
  char match[4][8]={"28F010","131072","28F020","262144"};
  intpos,
   mfg = 1,
   error = 0,
   field = 2;
  unsigned char*mfg_ptr,
         *id_ptr;

  // Process L type entries
  if (toupper(record[0]) == 'L')
  {
    // Get device type, memory size and byte position
    get_field(3,device_type,record);
    printf("Device programming = %s\n",device_type);
    get_field(6,mem_size,record);
    get_field(7,bytepos,record);
    pos=atoi(bytepos);
    last_address = atol(mem_size);
    printf("Last Address = %lX Hex\n",last_address);
    if (strcmp(device_type,match[0])==0)//if device=28f010
    { if (strcmp(mem_size,match[1])!=0)//check mem size = 131072
        error=INI_ERROR;
    }
    else if (strcmp(device_type,match[2])==0)//if device=28f020
    { if (strcmp(mem_size,match[3])!=0)//check mem size = 266,244
        error=INI_ERROR;
    }
    else
      error=INI_ERROR;//device type not in match[] array

    if (error)
    { sprintf(msg,"ERROR, Incorrect device type or memory size in pt2.ini\n");
      print_Message(msg);
    }

    // Get image filename(s)
    get_field(4,imgfile,record);
    ptr = strstr(imgfile,".");
    if (ptr != NULL)
      *ptr = 0x0;
    strcat(imgfile,".img");
    printf("infile = %s\n",imgfile);
    if (pos >= MAX_WIDTH)
      error=WIDTH_ERROR;
    strcpy(filename[pos],imgfile);
  }

  // Process M type entries
  if (toupper(record[0]) == 'M')
  { // Hexify MFG Codes and store in device and mfg code buffer
    mfg_ptr = mfg_code;
    id_ptr = device_code;
    while (get_field(field,code,record) != ERROR)  // Read all MFG Codes
    { if (mfg)
   *mfg_ptr++ = (unsigned char)strtol(code,&ptr,16);
      else
   *id_ptr++ = (unsigned char)strtol(code,&ptr,16);
      mfg = mfg ^ 1;
      field++;
    }
    // Count the number of codes
    code_count = 0;
    while (mfg_ptr-- != mfg_code)
      code_count++;
  }
  return(error);
}

//++++++++++ open data files function ++++++++++
//Function:  Opens the .img data file/s containing
//       the data for programing
//       device/s.
//      The file/s are pointed to by
//       a datafile pointer array.
//      This application only uses
//       one datafile pointer.
//Pre:none
//Post: If file/s not exist-> program exits.
// Otherwise:
//      1. datafile pointers -> to file/s
void
open_data_file(void)
{
  // Open data file
  datafile[0] = fopen(filename[0],"r+b");
  if (datafile[0] == 0)
  { sprintf(msg,"Error opening %s\n",filename[0]);
    print_Message(msg);
    cleanup();
    exit(FOPEN_ERROR);
  }
}

//++++++++++ init_dfp_cards function ++++++++++
//Function:  Initialize the CCC and DR2P cards
//       for this appliction.
//      Two cards are used
//       card 0 = address mode = address bus
//       card 1 = data mode = data bus +
//                        control bus
//
//Pre:   None.
//Post:  CCC + DR2P cards are initialized.
//++++++++++++++++++++++++++++++++++++++++++++++
void
init_dfp_cards()
{
  int n, addrsize;

  pt2init();     //DFP init
  initCard(0);   //card 0 init
  initCard(1);   //card 1 init
  releaseDR(0);  //this will clear all relays
  releaseDR(1);  //this will clear all relays
  setMode(0,ADDRESS);   //card 0 address mode
  ptEnable(0,ON);       //enable DFP, card 0
  setCountDir(0,UP);    //direction of count is up, card 0
  outpw(CARD0|PAB_CNTL,0xFF00 ); //Port A static out, Port B static out
  outp(CARD0|PC_CNTL,0x03 ); //Port C cntl, card 0

  setMode(1,DATA);//card 1, data mode
  ptEnable(1,ON);    //enable DFP, card 1
  outpw(DATA_DIR,INPUT);// Port A input, Port B input
  outp(CARD1|PC_CNTL,0x07 );// Port C control bits 0,1,2  output

  delay(0); // Calibrate delay
  if (last_address==0x40000)
    addrsize = 18;
  else if (last_address==0x20000)
    addrsize = 17;

  // Set relays for address
  for (n=0; n < addrsize; n++)
      outpw(CARD0|GA_INST,D_REED|nodA[n]|SET); // Address bus

  // Set relays for data
  for (n=0; n < 8; n++)
    outpw(CARD1|GA_INST,D_REED|nodA[n]|SET); // Data bus, card 1, Port A

  // Set relays for control
  outpw(CARD1|GA_INST,D_REED|NOD16|SET);  // WE*, card 1, Port C
  outpw(CARD1|GA_INST,D_REED|NOD17|SET);  // OE*, card 1, Port C
  outpw(CARD1|GA_INST,D_REED|NOD18|SET);  // CE*, card 1, Port C
  delay(50);// wait for relays, 50 ms

  return;
}

//++++++++++ ccc_timer_setup function ++++++++++
//Function:  Sets up delays using 8253
//      delay time = (count * .5us) + 3-5us software overhead
//
//      When dealing with delays in the low
//       micro-seconds(us), consideration must be
//       given for the time before the
//       programed delay can be triggered
//       and for the time after status
//       returns true before the next command
//       is given.
//      Best case is 1.5 us for each command,
//       therefore the time really aimed
//       for the 10us delay in this application
//       is 10us less two commands, or
//       10us - 3us= 7us.
//
//Pre:   CCC card is in DFP mode.
//Post:  card 0 : timer 0 : 10us delay
//       card 0 : timer 1 : 10ms delay
//++++++++++++++++++++++++++++++++++++++++++++++
void
ccc_timer_setup(void)
{
  // Set up timer 0 on CTC_CARD for 10us delay
  outp(CTC_CARD|CTC_MODE,0x32);
  outpw(CTC_CARD|CTC_CNT0,(unsigned int) 8);

  //Set up timer 1 on CTC_CARD for 10ms delay
  outp(CTC_CARD|CTC_MODE,0x72);
  outpw(CTC_CARD|CTC_CNT1,(unsigned int) 20000);
  return;
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
delay_7us (void)
{
    // delay 7us - wait for status bit true
    outp(CTC_CARD|TRG_0,0);//start delay
    while(!(readStatus(1) & 0x04));//poll for 7us delay done
    return;
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
delay_10ms (void)
{
    // delay 10ms - wait for status bit true
    outp(CTC_CARD|TRG_1,0);//start delay
    while(!(readStatus(1) & 0x08));//poll for 10ms delay done
    return;
}

//++++++++++ check_product_code function ++++++++++
//Function:  Checks for a match of the manufacturing id
// device code with the codes found in the pt2.ini file
//
//Pre:   CCC + DR2P cards initialized
//Post:  updates global variable failval
//       match: failval = 0
//       error: failval = 1
//++++++++++++++++++++++++++++++++++++++++++++++++
void
check_product_code(void)
{
  intidx,
   matched;
  unsigned char mfg_byte,
         id_byte;

  turn_on_VPP();//Vpp ON = 12V

  //Read Product Identification
  outpw(DATA_DIR,OUTPUT);// Port A output
  setCount(0,0x00,0x0000);// Autoselect Command Sequence
  write_byte (0x90);

  /* Read Product ID */
  outpw(DATA_DIR,INPUT); // Port A input
  mfg_byte = read_byte ();  // Read Manufactures Code
  outp( INC_ADDR, 0);
  id_byte = read_byte (); // Read Device Code
  printf("Manufacture Code = %2.2X, ",mfg_byte);
  printf("Device Code = %2.2X.\n",id_byte);
  // Check Manufacture Code and Device Code against PT2.INI 'M' Entry
  idx = 0;
  matched = 0;
  while (idx < code_count)
  {  if (mfg_code[idx] == mfg_byte && device_code[idx] == id_byte)
   matched = 1;
     idx++;
  }

  if (!matched)
  {  failval = 1;
     sprintf(msg,"Error, Manufacture or Device code does not match!!\n");
     print_Message(msg);
     sprintf(msg,"Manufacture Code = %2.2X\n",mfg_byte);
     print_Message(msg);
     sprintf(msg,"Device Code = %2.2X\n",id_byte);
     print_Message(msg);
  }

  reset_command();//Port A out-> Reset Command
  selVpp (1,OFF);// Disable Vpp
  return;
}

//++++++++++ check_erase function ++++++++++
//Function:  Sequentially checks all locations for
// the erased condition (0xFF) starting with
// the starting address thru to the
// ending address.  Return "error" representing
// chip is erased or not.
//
//Pre:   receives: long start_addr
//          long end_addr
//Post:  returns error:
//       error = NOERROR - device erased
//       error = ERROR - device NOT erased
//++++++++++++++++++++++++++++++++++++++++++++++
int
check_erase (long start_addr, long end_addr)
{
     unsigned int haddr,
           laddr;

     long         addr;

     haddr = (int)((start_addr >> 16) & 0x00FF);
     laddr = (int)(start_addr & 0xFFFF);
     setCount(0,haddr,laddr);// Card 0, addr high, addr low
     outpw(DATA_DIR,INPUT);// Port A input
     enable_read_byte ();
     for (addr = start_addr; addr <= end_addr; addr++)
     {  if (read_byte_only() != 0xFF)
        {  disable_read_byte ();
```

```
      return (ERROR);
   }
   outp( INC_ADDR, 0);// Increment Address
     }
     disable_read_byte ();
     return (NOERROR);
}

//++++++++++ erase_memory function ++++++++++
//Function:  1. Program all locations to zero.
//      2. Bulk erase (all locations to 0xFF).
//       The erase may be repeated 1000 times.
//
//Pre:   The device is not erased.
//Post:  failval = 0 - device succussfully erased
//  failval = 1 - device failed to erase.
//++++++++++++++++++++++++++++++++++++++++++++++
void
erase_memory (void)
{
     longaddr;
     unsigned inthaddr,
               laddr;
     int pulse_count;
     unsigned chardata_byte;


     turn_on_VPP();//Vpp ON = 12V

     // Program all locations to zero
     setCount(0,0x00,0x0000);// Card 0, addr high, addr low
     outpw(DATA_DIR,INPUT);// Port A input
     for (addr = 0; addr < last_address; addr++)
     {  if (read_byte() != 0x00)
        {  program_byte (0x00);
      outpw(DATA_DIR,OUTPUT);// Port A output
      write_byte (0x00); // Read command
      outpw(DATA_DIR,INPUT);// Port A input
   }
   outp( INC_ADDR, 0);// Increment Address
     }
     if (failval)
   return;

     // Erase Command Sequence
     setCount(0,0x00,0x0000);// Card 0, addr high, addr low
     pulse_count = 0;
     addr = 0x000000;
     while (pulse_count < 1000)
     {
        outpw(DATA_DIR,OUTPUT);// Port A output
        write_byte (0x20); // Erase Setup Command
   write_byte (0x20); // Erase Command
   delay_10ms ();
   while (addr < last_address)
   {  outpw(DATA_DIR,OUTPUT);// Port A output
      write_byte (0xA0); // Erase Verify Command

      outpw(DATA_DIR,INPUT);// Port A input
           outpw(DATA_DIR,INPUT);// dummy -for 6us delay

      data_byte = read_byte();
      if (data_byte != 0xFF)
      {   pulse_count++;
          break;
      }
      addr++;
      outp( INC_ADDR, 0);// Increment Address
   }
   if (addr == last_address)
   {  printf("\tTotal Erase count=%d\n",pulse_count);
      outpw(DATA_DIR,OUTPUT);// Port A output
      write_byte (0x00); // Read command
      return;
   }
     }

     failval = 1;// Exceeded pulse count limit
     sprintf(msg,"Erase pulse count EXCEEDED = %d!!\n",pulse_count);
     print_Message(msg);
     return;
}

//++++++++++ program_memory function ++++++++++
//Function:  1. Turn on VPP (12V).
//      2. Loads/increments an address location.
//      3. Gets a data byte via read_datafile
//       function.
//      4. Call the program_byte function.
//      5. Repeat steps 2,3,4 till all addresses
//       are programed or fails.
//      6. Turn off VPP, reset device, and return.
//
// note: program_count is used for debug to
//       check if device is trying more than
//       one write per address.
//
//Pre:   recieves: long start_addr
//Post:  failval = 0  - device successfully programmed
//  failval != 0 - device failed to program
//++++++++++++++++++++++++++++++++++++++++++++++
void
program_memory (long start_addr, long end_addr)
{
   long        addr;
   unsigned chardata_byte;
   unsigned inthaddr,
               laddr;
   program_count = 0;

   turn_on_VPP();//Vpp ON = 12V

   rewind (datafile[0]);// Return to the begining of data file
   addr = start_addr;
   haddr = (int)((addr >> 16) & 0x00FF);
   laddr = (int)(addr & 0xFFFF);
   setCount(0,haddr,laddr);// Card 0
   while (addr < end_addr && !failval)
   {
      read_datafile(datafile[0], &data_byte);  //Read next byte
      if (data_byte != 0xFF)
    program_byte (data_byte);
      addr++;
      outp(INC_ADDR, 0);// Increment Address
   }

   printf("\tTotal Program Count =%ld\n",program_count);
   return;
}

//++++++++++ program_byte function ++++++++++
//Function:  1. Tries to program a location up to
//       25 times.
//
//Pre:   Receives a data byte.
//Post:  failval = 0 - device successfully programmed
//  failval = 1 - device failed to program
//++++++++++++++++++++++++++++++++++++++++++++++
void
program_byte (unsigned char data)
{
   unsigned charverify_data;
   int         pulse_count = 0;

   if (failval)
     return;

   while (pulse_count < 25)
   {  outpw(DATA_DIR,OUTPUT);// Port A output

      write_byte (0x40); // Program Setup command
      write_byte (data); // Program data

      //delay 10us and verify command
      //(actually delay 7us, see ccc_timer_setup function)
      outp(CTC_CARD|TRG_0,0);//start delay
      outp(CNTL_BUS,CE_L | OE_H | WE_L);//address latched (wr=low)
      outp(DATA_BUS,0xC0);//Data = verify
      while(!(readStatus(1) & 0x08));//poll for delay done
      outp(CNTL_BUS,CE_L | OE_H | WE_H);//data latched (wr=high)
      outpw(DATA_DIR,INPUT);// Port A input
      outpw(DATA_DIR,INPUT);// dummy -for 6us delay

      verify_data = read_byte (); // Verify Data

      program_count++;
      if (verify_data == data)
    return;
      pulse_count++;
   }
   failval = 1;
   sprintf(msg,"Programming Pulse Count EXCEEDED=%d!!\n",pulse_count);
   print_Message(msg);
   return;
}

//++++++++++ read_datafile function ++++++++++
//Function:  Reads a data byte from  a file.
//       Exits if read error, otherwise
//       stores data byte.
//
//Pre:   Receives: pointer to a data file.
//            pointer to a data_byte variable.
//Post:  error -> exits with FREAD ERROR
//  noerror-> data pointed to by data_byte pointer
//++++++++++++++++++++++++++++++++++++++++++++++
void
read_datafile(FILE *datafile, unsigned char *data_byte)
{
   *data_byte = (unsigned char)(fgetc(datafile));   //Get data byte
   if (feof(datafile))
   {
      sprintf(msg,"Error reading data file\n");
      print_Message(msg);
      cleanup();
      exit (FREAD_ERROR);
   }
   return;
}

//++++++++++ verify_memory function ++++++++++
//Function:  Verifies all locations have
//       the correct data programmed.
//        Each location is compared with the
//       data file.
// The result is returned in the form of
// and error or noerror.
//
//Pre:   The device has been programmed.
//Post:  Return error:
//       error = NOERROR - device verified correctly
//       error = ERROR -   device not verify
//++++++++++++++++++++++++++++++++++++++++++++++
int
```

```
verify_memory(void)//datafile verification
{
   long  addr;
   unsigned char data_byte,
          expected_data;

   outpw(DATA_DIR,OUTPUT);// Port A output

   outpw(DATA_DIR,OUTPUT);// Port A output
   setCount(0,0x00,0x0000);// Read command sequence
   write_byte (0xFF);

   outpw(DATA_DIR,INPUT);// Port A input
   setCount(0,0,0);   //card 0, addr high, addr low, starting address

   //*********************************************
   //            Data Compare routine
   //*********************************************
   enable_read_byte ();
   rewind (datafile[0]);// Return to the begining of data file
   for (addr = 0; addr < last_address; addr++)    //address count
   {  read_datafile(datafile[0], &expected_data); //Read datafile
      data_byte = read_byte_only ();  // Read Memory
      if (data_byte != expected_data)//Compare datafile with memory
      {
        //printf("Address %lX not correct.",addr);
        //printf(" Read = %2X. Expected = %2X.\n",data_byte,expected_data);
        disable_read_byte ();
        return(ERROR);
      }
     outp( INC_ADDR, 0);   //increment address
   }
   disable_read_byte ();

  return(NOERROR);
}

//++++++++++ turn_on_VPP function ++++++++++
//Function:  Turns on the VPP to 12V.
//
//Pre:   None.
//Post:  VPP = 12V.
//++++++++++++++++++++++++++++++++++++++++++++
void
turn_on_VPP(void)
{
  /* Set Vpp to 12V */
  setVpp (1,240);//set the voltage for 12 V (.05 * 240=12)
  selVpp (1,ON);//turn Vpp ON
  delay (50);
}

//++++++++++ reset_command function ++++++++++
//Function:  1. Changes the Port A direction to output.
```

```
//      2. Writes the reset command 2 times.
//       Reset leaves the device in read mode and
//        safely aborts any operation.
//
//Pre:   None.
//Post:  device is reset, Port A output.
//++++++++++++++++++++++++++++++++++++++++++++++++
void
reset_command(void)
{
   outpw(DATA_DIR,OUTPUT);// Port A output
   write_byte (0xFF); // Reset Command
   write_byte (0xFF); // Reset Command
}

//++++++++++++++++++++++++++++++++++++++++++++++++
void
write_byte(unsigned char data)
{
    outp(CNTL_BUS,CE_L | OE_H | WE_L); //address latched
                           //     (write enable goes low)
    outp(DATA_BUS,data);// Data
    outp(CNTL_BUS,CE_L | OE_H | WE_H); //data  latched enable high
                           //     (write enable goes high)
    return;
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
enable_read_byte(void)
{
    outp(CNTL_BUS,CE_L | OE_L | WE_H);//enable read
    return;
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void
disable_read_byte(void)
{
    outp(CNTL_BUS,CE_L | OE_H | WE_H);//disable read
    return;
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
unsigned char
read_byte_only(void)
{
    int data_byte;

    data_byte = inp(DATA_BUS);//read data byte
    return ((unsigned char)data_byte);
}

// ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
unsigned char
read_byte(void)
{
    int data_byte;

    outp(CNTL_BUS,CE_L | OE_L | WE_H);//enable read
    data_byte = inp(DATA_BUS);//read data byte
    outp(CNTL_BUS,CE_L | OE_H | WE_H);//disable read
    return ((unsigned char)data_byte);
}

//+++++++++++ monitor function  +++++++++++++++++
//Function:  Displays the contents of DUT memory, 16 addresses
//       at a time, on the DFP terminal.
// The user enters enters an address 0 (zero) to
//       highest DUT address via the DFP keyboard.
//      The address are in hex -- example:  1c000
// The contents of 16 addresses of DUT memory will be displayed,
//       starting with the entered address.
//       example -- user enters:   300
//             the contents of address 300 hex (0x300)
//             is the first data displayed, followed by
//             the contents of the next 15 addresses.
//
// This is an extremely simple monitor.
// Error tolerance is high, and error handling is therefore nil.
// User cannot backspace. If he errs in typing the address -
//       just hit return, or the space bar, and try again.
// Entering an 'X' (exit) or 'Q' (quit) will terminate
//       the function.
// As long as no 'X' or 'Q' are recieved, the user may
//       continue to enter addresses to display the
//       Dut memory contents.
//
//Pre:Ptprog.exe is started from the command line
// with the starting argument -km or -pm.
// (see get_args function.)
//Post: The data contents of 16 DUT address locations are
// displayed, starting with the user input address.
// +++++++++++++++++++++++++++++++++++++++++++++++++++++
void
monitor(void)
{
  unsigned char t;// misc counting variable
  unsigned char byte;// location to store data
  unsigned long adr;// the "address" part of the query to DUT
  int count;// counts the bytes returned from DUT
  char cmd[80];// keyboard input string
  char done = 0;// a loop control item
  char *endptr;// pointer to end string for strtoul function

  selVpp (1,OFF);// Disable Vpp
  delay (50);
  outpw(DATA_DIR,INPUT);// Port A input
  while (!done)
  {
    printf("\n\aDUT Monitor.  Enter X to quit.\n");
    printf("->"); // prompt:
    scanf("%s",cmd);

    // Convert input string to upper case.
    for (count = 0; count < 6 ; count++)
      cmd[count] = (char)toupper(cmd[count]);

    // These are the exit criteria: x, X, q, or Q.
    if ( (cmd[0] == 'X') || (cmd[0] == 'Q'))
    {
      done = 1;
      printf("\aExiting monitor\n");
      break;
    }

    //convert ascii string into unsigned long (hex= base 16)
    adr = strtoul(cmd, &endptr, 16);
    printf("adr: %05.5lX\n",adr); // verify address.

    // read 16 bytes
    setCount(0,(int)((adr >> 16) & 0xFF),(int)(adr & 0xFFFF));
    for (count = 0 ; count < 16 ; count++)
    {
      // get one byte per pass thru this loop
      byte=read_byte();
      outp( INC_ADDR, 0);
      printf("%02.2X ",byte);
    }
  } // end of while !done
} // end of Monitor

//++++++++++ cleanup function ++++++++++++++
//Function:  Release the DR2P cards from
// DFP control, close all files+ handles.
//
//Pre:Program is about to exit (return).
//Post: DFP releases DR2P control, all
// files + handles closed.
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++
void
cleanup(void)
{
  releaseDR(0);// Disable PrompTest - card 0
  releaseDR(1);// Disable PrompTest - card 1
  close(handle);// Close com port handle
  fcloseall();// Close any open files
  return;
}
```

```
//++++++++++ process message function ++++++++++++++
//Function:  print failure/error message
//       to both DFP and 18xx screen
//
//Pre:Receives msg buffer with string.
//Post: Prints message to DFP screen.
// Sends message to 18xx to printed to 18xx screen.
//
//+++++++++++++++++++++++++++++++++++++++++++++++++++
void
print_Message(char *msg)
{
   printf("%s",msg); //print to DFP screen
   send18xxMsg(handle,DFP_18XX_MSG,msg); //send to 18xx DigFuncProc
                                 //worksheet
  return;
}
```

# PT2.H Listing 7

A listing of the contents of the PT2.H file appears on the following pages.

```
/*
Pt2.h for DFP
Digital Function Processor system Copyright (C) 1993,1994 Teradyne Inc.

Description-
      Pt2.h header file for DFP
*/
#define VERSION     “B0_1_1_46262”
#define PTPROG“PTPROG”

#define MAX_COM_SIZ600
#define MAX_L_REC 16
#define I_REC_SIZ128
#define D_REC_SIZ128
#define MAX_FIELD 64
#define MAX_REC_SIZ128
#define CFG_SIZ 60

#ifndef TRUE
#define TRUE  1
#define FALSE 0
#endif

#ifndef ERROR
#define ERROR (-1)
#endif

#ifndef NOERROR
#define NOERROR 0
#endif

#define DONT_SAV_CR0
#define SAV_CR1

#define TIMEOUTERR       (-2)
#define ESC  27
#define EVER  ;;

#define MS_PER_TICK      55

#ifdef tolower
#undef tolower
#endif

#ifdef toupper
#undef toupper
#endif

#ifdef inp
#undef inp
#endif

#ifdef inpw
#undef inpw
#endif

#ifdef outp
#undef outp
```

```
#endif

#ifdef outpw
#undef outpw
#endif

// Used by routines: initDirPath(), SaveDirPath() and restoreDirPath().
// Note: This structure must be initailized by initDirPath() before it is used!
struct dirPath
{
    int  drive;   // drive number: 1 = A, 2 = B, 3 = C ...
    char path[_MAX_DIR];// a directory path in ‘drive’
};

/*        CCC defines            */

#define OFF 0
#define ON  1

#define DOWN 0
#define UP   1

                  /* Card modes */
#define PARALLEL 0
#define SERIAL   1
#defineADDRESS  2
#defineDATA     3

                  /* Card Addresses */
#define CARD0     0x0380
#define CARD1     0x0388
#define CARD2     0x0390
#define CARD3     0x0398
#define CARD4     0x03A0

                  /* Block 0 - Address Counter */
#define ADD_R0    0x0000
#define ADD_R1    0x0002
#define ADD_R2    0x0004
#define ADD_INC   0x0006
                  /* Block 1 - Port Control */
#define PC_CNTL  0x1000
#define PAB_CNTL  0x1002
#define PA_HSHK   0x1004
#define PB_HSHK   0x1006
                  /* Block 2 - Gate Array Control */
#define GA_INST  0x2000
#define GA_ENAB   0x2002
#define GA_DATA   0x2004
#define GA_GRAY   0x2006
                  /* Block 3 - Port Data */
#define PA_DATA   0x3000
#define PB_DATA   0x3002
#define PC_DATA   0x3004
#define PAB_DATA  0x3006
                  /* Block 4 - CTC */
#define CTC_CNT0  0x4000
```

```
#define CTC_CNT2  0x4002
#define CTC_CNT1  0x4004
#define CTC_MODE  0x4006
                  /* Block 5 - SIO */
#define SIO_BCOM  0x5000
#define SIO_BDAT  0x5002
#define SIO_ACOM  0x5004
#define SIO_ADAT  0x5006
                  /* Block 6 - Misc */
#define STATUS    0x6000
#define RST       0x6002
#define VPP       0x6004
#define MISC      0x6006
                  /* Block 7 - Timer Trigger */
#define TRG_0     0x7000
#define TRG_1     0x7002
#define TRG_2     0x7004

#define PA   0    /* Port A */
#define PB   1    /* Port B */
#define PAB  2    /* Port A and B combined 16-bit port */

/*        Driver Receiver defines            */

#ifdef SET
#undef SET
#endif
#define SET       0x1
#define CLR       0x0

                  /* Relays */
#define V_REED    0x360
#define D_REED    0x8
#define E_REED    0xA
#define F_REED    0xC
#define G_REED    0xE

#define MAST_ED   0x120
#define MAST_FD   0x128
#define MAST_GD   0x220
#define MAST_ES   0x122
#define MAST_FS   0x12A
#define MAST_GS   0x222
#define MAST_EBR  0x228
#define MAST_FBR  0x22A
#define MAST_GBR  0x22C
#define MAST_DED  0x124
#define MAST_DFD  0x12C
#define MAST_DGD  0x224
#define MAST_DES  0x126
#define MAST_DFS  0x12E
#define MAST_DGS  0x226

                  /* Node addresses */
#define NOD0      0x0020
```

```
#define NOD1      0x0060
#define NOD2      0x00A0
#define NOD3      0x00E0
#define NOD4      0x1020
#define NOD5      0x1060
#define NOD6      0x10A0
#define NOD7      0x10E0
#define NOD8      0x2020
#define NOD9      0x2060
#define NOD10     0x20A0
#define NOD11     0x20E0
#define NOD12     0x3020
#define NOD13     0x3060
#define NOD14     0x30A0
#define NOD15     0x30E0
#define NOD16     0x4020
#define NOD17     0x4060
#define NOD18     0x40A0
#define NOD19     0x40E0
#define NOD20     0x5020
#define NOD21     0x5060
#define NOD22     0x50A0
#define NOD23     0x50E0
#define NOD24     0x6020
#define NOD25     0x6060
#define NOD26     0x60A0
#define NOD27     0x60E0
#define NOD28     0x7020
#define NOD29     0x7060
#define NOD30     0x70A0
#define NOD31     0x70E0

/*--------------------- async ---------------------*/
#define NOCHANGE-1

#define BAUD110         0       /* 110 baud */
#define BAUD150         1       /* 150 baud */
#define BAUD300         2       /* 300 baud */
#define BAUD600         3       /* 600 baud */
#define BAUD1200        4       /* 1200 baud */
#define BAUD2400        5       /* 2400 baud */
#define BAUD4800        6       /* 4800 baud */
#define BAUD9600        7       /* 9600 baud */
#define BAUD19200       8       /* 19200 baud */
#define BAUD38400       9       /* 38400 baud */
#define BAUD57600       10      /* 57600 baud */
#define BAUD115200      11      /* 115.2 kbaud */


/*=====Parities                                          */
#define PARITY_NONE     0       /* No parity */
#define PARITY_ODD      1       /* ODD parity */
#define PARITY_EVEN     2       /* Even parity */
#define PARITY_SODD     3       /* Sticky ODD parity */
#define PARITY_SEVEN    4       /* Sticky Even parity */

/*=====Lengths                                           */
#define LENGTH_5        0       /* 5 bits */
```

```
#define LENGTH_6        1       /* 6 bits */
#define LENGTH_7        2       /* 7 bits */
#define LENGTH_8        3       /* 8 bits */

/*=====Stobits                                                  */
#define STOPBIT_1       0       /* 1 Stop bit */
#define STOPBIT_2       1       /* 2 Stop bit */

/*=====PROTOCOL                                                 */
#define PROT_NONE       0       /* No X  protocol/Hardware protocol */
#define PROT_XRCV       1      /* X protocol on reception/Hardware protocol */
#define PROT_XXMT       2       /* X protocol on xmission/Hardware protocol */
#define PROT_XALL       3       /* X protcol on XMIT&RCV/Hardware protocol */
#define PROT_HNONE      4       /* No X protocol/No Hardware protocol */
#define PROT_HXRCV      5      /* X protocol on reception/No hardware prtocl */
#define PROT_HXXMT      6      /* X protocol on xmission/No hardware protocol*/
#define PROT_HXALL      7      /* X protocol on XMIT&RCV/No hardware protocol*/

#define INPUT_FLUSH0
#define OUTPUT_FLUSH1
#define IO_FLUSH2

struct port_status
{
    unsigned shortBaud;// according to the BAUD defines
    unsigned shortParity;// according to the Parity defines
    unsigned shortLenght;// according to the Length defines
    unsigned shortStopBits;// According to the Stop bit defines
    unsigned shortProtocol;// according to the Protocol defines
    unsigned shortInBufLen;// Input buffer lenght
    unsigned shortOutBufLen;// Output buffer lenght
    unsigned shortInBufCnt;// number of characters in the input
    unsigned shortOutBufCnt;// # of charaters in the outbut buffer
    unsigned charInWait;// block time for input in 1/18 sec
    unsigned charOutWait;// block time for output in 1/18 sec
    unsigned charModemStatReg;
    unsigned charLineStatReg;
};


/*        Slave error codes      */

#define DATE_SET             20  // Date and time set successfully
#define P_DONE               21  // Ptprog finished
#define TERMINATE            22  // Slave program terminating
#define SLAVE_OK             23  // Slave acknowledge
#define TIME_ERROR           24  // Error setting time
#define DATE_ERROR           25  // Error setting date
#define XLATE_ERROR          26  // Error starting xlate program
#define PTPROG_ERROR         27  // Error starting ptprog program
#define INVALID_COMMAND      28  // Invalid command recieved
#define INVALID_DRIVE        29  // Invalid drive specified in Job path
#define INVALID_VERSION      30  // Unable to create module directory
#define INVALID_LRECORD      31  // Valid records are 1 through MAX_L_REC
#define DSZ_ERROR            32  // Error starting zmodem
#define INVALID_JOBPATH      33  // Invalid job path specified

/*        Pcom error codes       */
```

```
#define PCOM_ARG_ERR         40  // Wrong number of arguments to pcom
#define PCOM_PRT_ERR         41  // Error opening serial port
#define PCOM_TIMEOUT         42  // Serial port timeout
#define PCOM_RD_ERR          43  // Serial port read error
#define PCOM_CFG_ERR         44  // Unable to find pcom.cfg file

/*        Ptver error codes      */

#define PVER_DIRPATH_ERR     58  // Internal error in a dirPath routine
#define PVER_NO_HOOK         59  // Failed to hook to pcio
#define PVER_ARG_ERR         60  // Invalid comport designation
#define PVER_ARGC_ERR        61  // Wrong number of arguments
#define PVER_PRT_ERR         62  // Error opening com port
#define PVER_VER_ERR         63  // Slave and Ptver software version mismatch
#define PVER_TIMEOUT         64  // Serial port timeout
#define INVALID_18XX_DRIVE   65  // Drive nonexistant on DFP
#define INVALID_18XX_PATH    66  // Path nonexistant on DFP
#define REBOOT_PT2_ERR       67  // Unable to start Ptboot
#define CFG_ERROR            68  // Unable to create pcom.cfg
#define CFG_MISMATCH         70  // pt2.cfg differs from pt2.ini file

/*        Ptprog (DFP to 18xx) message codes     */

#define DFP_18XX_MSG         73  // Send a DFP msg to 18xx
#define DFP_18XX_ERR_MSG     74  // Send a DFP error msg to 18xx, to be
                         //   displayed on the 18xx screen in a red box.

/*        Ptprog error codes     */

#define PT_CFG_ERROR         77  // Configuration error
#define ALGORITHM_ERROR     78  // Multiple algorithms declared in pt2.ini file
#define INI_FIELD_ERROR      79  // Not enough fields in pt2.ini file
#define PTPROG_PRT_ERROR     80  // Error opening com port
#define PTPROG_PARAM_ERROR   81  // Error setting port parameters
#define PTPROG_ALIVE         82  // Ptprog is working
#define PASS                 83  // Device(s) passed programming
#define FAIL                 84  // Device(s) failed programming
#define FOPEN_ERROR          85  // Error opening an .img file
#define FREAD_ERROR          86  // Error reading an .img file
#define INI_ERROR            87  // Error reading or opening .ini file
#define WIDTH_ERROR          88  // Bus position larger than allowed
#define DEV_NOT_FOUND        89  // Programming algorithm does not exist

/*        Xlate error codes      */

#define XLATE_DONE           90  // Translate complete
#define XLATE_ALIVE          91  // Xlate working
#define INIFILE_ERROR        92  // Error opening pt2.ini file
#define IMGFILE_ERROR        93  // Error opening .img file
#define SFILE_ERROR          94  // Error opening S-record file
#define FIELD_ERROR          95  // Error in number of fields in pt2.ini
#define CHKSUM_ERROR         96  // Error in S-record checksum
#define DEVICE_FIL_ERROR     97  // Error opening device.dat file
#define DEVICE_ERROR         98  // Device not found in device.dat file
#define INVALID_RECORD       99  // Unknown record type
#define ADDRESS_ERROR       100  // S-record too large for device

/*        Algorithm error codes  */
```

```
#define ALG_PRT_ERROR       110  // Error opening com port
#define ALG_PARAM_ERROR     111  // Error setting port parameters
#define ALG_CFG_ERROR       112  // Configuration error
#define ALG_INI_ERROR       113  // Error opening/reading pt2.ini file
#define ALG_NOD_ERROR       114  // Node outside DFP configuration
#define ALG_STAT_ERROR      115  // Invalid state requested
#define ALG_MEM_ERROR       116  // Memory alocation error
#define ALG_WIDTH_ERR       117  // Number of devices does not match algorithm
#define NOD_CONFLICT        118  // Node already used
#define INIT_ERROR          119  // CCC not responding

/*       Global Variables  */
extern int nodA[32];            /* Node addresses */
extern int ca[5];/* Card addresses */
extern int miscReg[5];/* Misc reg image */


int
write_port (int fd, char *port_st);
      /*
       * Procedure:   write_port
       *
       * Function:    String is written to serial port fd
       *
       * Parameters:
       *              fd - int - serial port file descriptor
       *              port_st - *char - pointer to string
       *
       * Return Values:
       *              ERROR, TIMEOUTERR, NOERROR - if the port was written
       *                                           successfully
       *
       * Discussion:
       *              if the port cannot be written to return ERROR
       *              if the port times out return TIMEOUTERR
       *              the status of the port is read
       *          if the buffer is not full characters are written
       *              until a terminating null or
       *              if the string is empty
       *              return NOERROR
       *
       * Imports modified:
       *              NONE
       */

int
read_port (int fd, char *port_st, int wait);
```

```
      /*
       * Procedure:   read_port
       *
       * Function:    Serial port is read until Line Feed is encountered.
       *          Note: Carriage Returns are not saved in port_st[].
       *
       * Parameters:
       *              fd - int - serial port file descriptor
       *              port_st - *char - pointer to string to be modified
       *              wait - int - time in seconds to wait for string
       *
       * Return Values:
       *          ESC         - ESC was pressed
       *          ERROR       - read() returned -1: bad file handle
       *          TIMEOUTERR- have not received CR-LF within
       *                          'wait' seconds
       *          NOERROR     - no errors, port was read successfully
       *
       * Discussion:
       *              if the port cannot be read return ERROR
       *              if wait = 0 then wait forever else wait specified
       *              seconds for string then return TIMEOUTERR
       *              if the port times out return TIMEOUTERR
       *              the status of the port is read
       *          if there are characters in the buffer the port is read
       *              until a terminating line feed or
       *              if the buffer is empty an empty string is returned
       *              return NOERROR
       *
       * Imports modified:
       *              port_st
       */

int
really_read_port (int fd, char *port_st, int wait, int savCR);

      /*
       * Procedure:really_read_port
       *
       * Function:Same as read_port() except Carriage Returns can be
       *          are saved in port_st[].
       *
       * Parameters:fd- same as in read_port()
       *              port_st - same as in read_port()
       *              wait- same as in read_port()
       *          savCR - set to SAV_CR if you want save carriage
       *                    returns, set to DONT_SAV_CR if you do not
       *                    want carriage returns.
       */

int
isInBufferEmpty(int fd);
```

```
      /*
       * Procedure:   isInBufferEmpty
       *
       * Function:    check to see if the serial port input buffer has
       *              character(s) to be read
       *
       * Parameters:
       *              fd - int - serial port file descriptor
       *
       * Return Values:
       *              ERROR, FALSE, TRUE - if the buffer is empty
       *
       * Discussion:
       *              the status of the port is read
       *          if there are characters in the buffer return FALSE
       *          else return TRUE
       *
       * Imports modified:
       *              NONE
       */

int
crlf_port(int fd);

      /*
       * Procedure:   crlf_port
       *
       * Function:    Carriage return is written to serial port fd
       *
       * Parameters:
       *              fd - int - serial port file descriptor
       *
       * Return Values:
       *              ERROR, TIMEOUTERR, NOERROR - if the port was written
       *                                           successfully
       *
       * Discussion:
       *              if the port cannot be written to return ERROR
       *              if the port times out return TIMEOUTERR
       *              the status of the port is read
       *          if the buffer is not full carriage return is written
       *              return NOERROR
       *
       * Imports modified:
       *              NONE
       */

int
md_cd(char *path);
```

```
      /*
       * Procedure:   md_cd
       *
       * Function:    creates and changes to path directory
       *
       * Parameters:
       *              *path - char - pathname
       *
       * Return Values:
       *              ERROR, NOERROR - if directory successfully created
       *                               and changed to.
       *
       * Discussion:
       *              if unable to create or change to directory return ERROR
       *              else return NOERROR
       *
       * Imports modified:
       *              NONE
       */

int
get_dir(int fn, char *outstr, char *instr);

      /*
       * Procedure:   get_dir
       *
       * Function:    extracts directory name specified
       *              by field number from path
       *
       * Parameters:
       *              fn - int - field number
       *              *outstr - char - returned directory
       *              *instr - char - path
       *
       * Return Values:
       *              ERROR, NOERROR - if valid directory name returned
       *
       * Discussion:
       *              if invalid directory name return ERROR
       *              else return NOERROR
       *
       * Imports modified:
       *              outstr
       */

#     ifndef TURBO
unsigned long
delay(unsigned long ms);
#     endif

      /*
       * Procedure:   delay
       *
       * Function:    delays for specified milliseconds
       *
       * Parameters:
       *              ms - int - number of milliseconds
       *
       * Return Values:
       *              NONE
       *
       * Discussion:
       *              delays for period specified by ms
       *
       * Imports modified:
       *              NONE
       */

void
sync(int fd);

      /*
       * Procedure:   sync
       *
       * Function:    syncronizes DFP and 18xx computers
       *
       * Parameters:
       *              fd - int - file descriptor
       *
       * Return Values:
       *              NONE
       *
       * Discussion:
       *              Ports trade characters (H) until both sides have
       *              seen 25 characters. A terminating character is
       *              then sent. The ports are then read until the
       *              terminating character (I) is found.
       *
       * Imports modified:
       *              NONE
       */

char **
breakUp(char *str);
```

```
      /*
       * Procedure:   breakUp
       *
       * Function:    returns array of pointers
       *
       * Parameters:
       *              *instr - char - string of arguments to seperate
       *
       * Return Values:
       *              char **Array of null terminated strings.
       *                Null pointer signifies end of array.
       *
       * Discussion:  breakUp takes a string of arguments seperated by a
       *              space and returns an array of pointers to null
       *              terminated strings that contain the arguments. The
       *              end of the argument list is signified by a null pointer.
       *
       * Imports modified:
       *              NONE
       */

unsigned int
hex(char input);

      /*
       * Procedure:   hex
       *
       * Function:    returns integer value of ascii hex character
       *
       * Parameters:
       *              input - char - character to convert
       *
       * Return Values:
       *              int               Converted value of character.
       *
       * Discussion:  Hex takes an ascii hex character and converts to
       *              the integer value.
       *
       * Imports modified:
       *              NONE
       */

void
keep_alive(int fd, int error);
```

```
      /*
       * Procedure:   keep_alive
       *
       * Function:    sends ascii string representing an integer to com port
       *
       * Parameters:
       *              fd - int - file descriptor
       *              error - int - integer to send
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Converts error to ASCII string and sends to com port
       *              specified by fd.
       *
       * Imports modified:
       *              NONE
       */

int
get_field(int fn, char *outstr, char *instr);

      /*
       * Procedure:   get_field
       *
       * Function:    returns requested field from input string
       *
       * Parameters:
       *              fn - int - field number
       *              *outstr - char - output string
       *              *instr - char - input string
       *
       * Return Values:
       *              ERROR, NOERROR - if valid field returned
       *
       * Discussion:  Returns a string (outstr) specified by field number (fn)
       *              from an input string (instr) which consists of multiple
       *              fields seperated by commas. First field is field 1.
       *
       * Imports modified:
       *              NONE
       */

/************************* CCC prototypes *************************/
```

```
                         /* SIO Functions */

void
sioComWr(int card, int port, int reg, int data);

      /*
       * Procedure:   sioComWr
       *
       * Function:    Write to SIO register
       *
       * Parameters:
       *              card - int - CCC number
       *              port - int - port address
       *              reg - int - register number
       *              data - int - data to write
       *
       * Return Values:
       *              NONE
       *
       * Discussion:
       *              This routine writes data to the SIO write
       *              register specified by reg. Port is defined as
       *              SIO_ACOM or SIO_BCOM.
       *
       * Imports modified:
       *              NONE
       */

int
sioComRd(int card, int port, int reg);

      /*
       * Procedure:   sioComRd
       *
       * Function:    Read SIO register
       *
       * Parameters:
       *              card - int - CCC number
       *              port - int - port address
       *              reg - int - register number
       *
       * Return Values:
       *              int        register data
       *
       * Discussion:
       *              This routine returns the contents of the SIO read
       *              register specified by reg. Port is defined as
       *              SIO_ACOM or SIO_BCOM.
       *
       * Imports modified:
       *              NONE
       */

void
sioWr(int port, unsigned char ch);
```

```
      /*
       * Procedure:   sioWr
       *
       * Function:    Transmit character to the DUT
       *
       * Parameters:
       *              port - int - port address
       *              ch - unsigned char - character to send
       *
       * Return Values:
       *              NONE
       *
       * Discussion:
       *              This routine transmits a character out one of the
       *              CCC serial ports. Port address is the combined value
       *              of the CCC address and either SIO_ADAT or SIO_BDAT.
       *
       * Imports modified:
       *              NONE
       */

char
sioRd(int port);

      /*
       * Procedure:   sioRd
       *
       * Function:    Return a character from the DUT.
       *
       * Parameters:
       *              port - int - port address
       *
       * Return Values:
       *              int        character read
       *
       * Discussion:
       *              This routine receives a character from one of the
       *              CCC serial ports. The calling program must test
       *              sioRdrf(). This function does not, for fear of getting
       *              trapped forever by a dead DUT. Calling this function
       *              will clear rdrf. Port address is the combined value
       *              of the CCC address and either SIO_ADAT or SIO_BDAT.
       *
       * Imports modified:
       *              NONE
       */

int
sioRdrf(int port);
```

```
      /*
       * Procedure:   sioRdrf
       *
       * Function:    Test receiver data ready flag
       *
       * Parameters:
       *              port - int - port address
       *
       * Return Values:
       *              int        state of rdrf bit
       *
       * Discussion:
       *              This routine returns the state of the receiver data
       *              ready flag. Return 1 if character is available 0 if
       *              not. Port address is the combined value of the CCC
       *              address and either SIO_ADAT or SIO_BDAT.
       *
       * Imports modified:
       *              NONE
       */

/************************ Initialization Functions ************************/

void
pt2init(void);

      /*
       * Procedure:   pt2init
       *
       * Function:    Initializes card and node arrays
       *
       * Parameters:
       *              NONE
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  pt2init initializes the ca[] (card address) and
       *              nodA[] (node address) arrays to the values defined
       *              in the pt2.h file.
       *
       * Imports modified:
       *              NONE
       */

int
initCard(int card);
```

```
      /*
       * Procedure:   initCard
       *
       * Function:    initializes channel control card
       *
       * Parameters:
       *              card - int - CCC number
       *
       * Return Values:
       *              NOERROR, ERROR if card doesn’t respond
       *
       * Discussion:  Performs reset on CCC. Registers cleared, VPP off,
       *              timers set to mode 1, SIO channels reset.
       *
       * Imports modified:
       *              NONE
       */

void
setMode(int card, int mode);

      /*
       * Procedure:   setMode
       *
       * Function:    initializes channel control card
       *
       * Parameters:
       *              card - int - CCC number
       *              mode - int - mode to set CCC card to
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Sets specified card to one of four modes.
       *              SERIAL   - port C will have serial functions.
       *              PARALLEL - port C will have handshake functions.
       *              DATA     - port C will have data io functions.
       *              ADDRESS  - port C will have address generator functions.
       *
       * Imports modified:
       *              miscReg[card]
       */

void
ptEnable(int card, int state);
```

```
      /*
       * Procedure:   ptEnable
       *
       * Function:    enables channel control card
       *
       * Parameters:
       *              card - int - CCC number
       *              state - int - state to set CCC card to
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Enables/disables Channel Control Card/DR2p card
       *              If state is ON then channel card has control of
       *              driver receiver card.
       *              If state is OFF then driver receiver card is back
       *              under 18xx control.
       *
       * Imports modified:
       *              miscReg[card]
       */

void
rstDR(int card);

      /*
       * Procedure:   rstDR
       *
       * Function:    resets driver receiver card
       *
       * Parameters:
       *              card - int - CCC number attached to DR2p
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Resets driver receiver card. Resets all gate array
       *          latches clears all relays (no disables no vreeds),
       *          clears all memory (no disables), clears last digital
       *          driver state, and clears eight direct connect relays.
       *          CCC card must be enabled.
       *
       * Imports modified:
       *              NONE
       */

void
releaseDR(int card);
```

```
      /*
       * Procedure:   releaseDR
       *
       * Function:    clears relays on driver receiver card
       *
       * Parameters:
       *              card - int - CCC number attached to DR2p
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Clears all relays (no disables no vreeds), and clears
       *              eight direct connect relays. CCC card does not have
       *              to be enabled.
       *
       * Imports modified:
       *              NONE
       */

/**************************** VPP Functions ****************************/

void
setVpp(int card, int volts);

      /*
       * Procedure:   setVpp
       *
       * Function:    sets vpp voltage
       *
       * Parameters:
       *              card - int - CCC number
       *              volts - int - voltage to set
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Programs the Vpp voltage for the Channel Control
       *              card. Volts is in 50mv increments. 0-255 where
       *              255 = 12.75 volts.
       *
       * Imports modified:
       *              NONE
       */

void
selVpp(int card, int state);
```

```
      /*
       * Procedure:   selVpp
       *
       * Function:    turns on/off vpp
       *
       * Parameters:
       *              card - int - CCC number
       *              state - int - state to set vpp
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Turns on/off vpp voltage.
       *              If state is ON then vpp is on.
       *              If state is OFF then vpp is off.
       *
       * Imports modified:
       *              miscReg[card]
       */

int
measVpp(int card);

      /*
       * Procedure:   measVpp
       *
       * Function:    measure DUT vpp
       *
       * Parameters:
       *              card - int - CCC number
       *
       * Return Values:
       *              int        Voltage measured in 50mv increments.
       *
       * Discussion:  Measure DUT VPP by successive approximation A-D
       *
       * Imports modified:
       *              miscReg[card]
       */

/*********************** Address Counter Functions ***********************/
```

```
void
setCountDir(int card, int dir);

      /*
       * Procedure:   setCountDir
       *
       * Function:    set direction of address counted
       *
       * Parameters:
       *              card - int - CCC number
       *              dir  - int - direction
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Set direction of address counter to either
       *              UP or DOWN.
       *
       * Imports modified:
       *              miscReg[card]
       */

void
incCount(int card);

      /*
       * Procedure:   incCount
       *
       * Function:    increment address counter
       *
       * Parameters:
       *              card - int - CCC number
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Increment/decrement address counter by 1.
       *
       * Imports modified:
       *              NONE
       */

void
setCount(int card, int addh, int addl);
```

```
      /*
       * Procedure:   setCount
       *
       * Function:    set address counter
       *
       * Parameters:
       *              card - int - CCC number
       *              addl - int - least significant 16 bits
       *              addh - int - most significant 8 bits
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Set 24 bit address counter to combined value
       *              of addh and addl.
       *
       * Imports modified:
       *              NONE
       */

void
getCount(int card, int *addh, int *addl);

      /*
       * Procedure:   getCount
       *
       * Function:    Get current address counter
       *
       * Parameters:
       *              card  - int - CCC number
       *              *addl - int - least significant 16 bits
       *              *addh - int - most significant 8 bits
       *
       * Return Values:
       *              NONE
       *
       * Discussion:  Get current value of 24 bit address counter.
       *
       * Imports modified:
       *              *addl
       *              *addh
       */

/*********************** Parallel Port Functions ***********************/
```

```
int
getPortData(int card, int port);

      /*
       * Procedure:   getPortData
       *
       * Function:    Read data from port
       *
       * Parameters:
       *              card - int - CCC number
       *              port - int - port to read from
       *
       * Return Values:
       *              int        data read from port
       *
       * Discussion:
       *              The port functions all work with a port argument,
       *          which is defined as follows:-
       *
       *              PA  = Port A  8 bits D0-7.
       *              PB  = Port B  8 bits D8-15.
       *              PAB = Port A+B  16 bits D0-15.
       *
       *              The ports must have been initialised into the correct
       *          mode prior to these commands using the setMode()
       *          function.
       *
       * Imports modified:
       *              NONE
       */

int
readStatus(int card);

      /*
       * Procedure:   readStatus
       *
       * Function:    Read status register
       *
       * Parameters:
       *              card - int - CCC number
       *
       * Return Values:
       *              int        contents of the status register
       *
       * Discussion:
       *              returns the contents of the CCC status register
       *
       * Imports modified:
       *              NONE
       */

void
setPortData(int card, int port, int data);
```

```
      /*
       * Procedure:   setPortData
       *
       * Function:    Write data to port
       *
       * Parameters:
       *              card - int - CCC number
       *              port - int - port to write to
       *              data - int - data to write
       *
       * Return Values:
       *              NONE
       *
       * Discussion:
       *              The port functions all work with a port argument,
       *              which is defined as follows:-
       *
       *              PA  = Port A  8 bits D0-7.
       *              PB  = Port B  8 bits D8-15.
       *              PAB = Port A+B  16 bits D0-15.
       *
       *              The ports must have been initialised into the correct
       *          mode prior to these commands using the setMode()
       *          function.
       *
       * Imports modified:
       *              NONE
       */

/**************************** async functions ****************************/

int
serial_get(int handle, struct port_status *P_stat);

      /*
       * Procedure:serial_get
       *
       * Function:Function to get communication paramters
       *
       * Inputs:  handle: The file handle returned from an open.
       *                P_stat: The address of a structure port_status.
       *
       * Return Value:False if operantion could not be provided,
       *                otherwise true.
       */

int
serial_set (int handle, int baud, int par, int len, int stp, int proto, int
       i_wait, int o_wait);

      /*
       * Procedure:serial_set
       *
       * Function:Function to set communication paramters
       *
       * Inputs:  handle: The file handle returned from an open.
       *                baud :  The baud rate according to the BAUDxxx
       *                        manifest defines
       *                par:  The parity according to the PARITYxxx
       *                        defines.
       *                len:  The byte lenght according to the
       *                        LENGHTxxxx defines.
       *                stp:  The number of stoppbits according to
       *                        the STOPPBIT def.
       *                proto:The type of handshake according to the
       *                        PROTOxxx def.
       *                i_wait:The input timeout in milli sec. up to
       *                        14 seconds.
       *                o_wait:The output timeout in milli sec. up to
       *                        14 seconds.
       *
       * Return Value:False if operantion could not be provided,
       *                otherwise true.
       *
       * Note:    All of the above arguments, with exception of
       *                'handle' can have the define NOCHANGE if only
       *                a partial setup is required.  The i_wait is
       *                the time a read() will be suspened for if there
       *                is no input to the port.  The o_wait is the
       *                time is the time that is waited for space in
       *                the output buffer to become availabale while
       *                in write()
       */

int
serial_wait_tx_empty (int handle);

      /*
       * Procedure:serial_wait_tx_empty
       *
       * Function:Function to wait for the transmit buffer to
       *                empty out
       *
       * Inputs:  handle: The file handle returned from an open.
       *
       * Return Value:False if operantion could not be provided,
       *                otherwise true.
       */

int
serial_flush_buff (int handle, int input_output);
```

```
      /*
       * Procedure:serial_flush_buff
       *
       * Function:Function to truncate the input- and/or
       *               output- buffers
       *
       * Inputs:  handle: The file handle returned from an open.
       *               nput_output: one of the xxxxFLUSH defines
       *
       * Return Value:False if operantion could not be provided,
       *               otherwise true.
       */

int
serial_DTR (int handle, int on_off);

      /*
       * Procedure:serial_DTR
       *
       * Function:Function to manipulate the DTR line manually
       *
       * Inputs:  handle: The file handle returned from an open.
       *               on_off: A 1 for on, 0 for off
       *
       * Return Value:False if operantion could not be provided,
       *               otherwise true.
       */

int
serial_RTS (int handle, int on_off);

      /*
       * Procedure:serial_RTS
       *
       * Function:Function to manipulate the RTS line manually
       *
       * Inputs:  handle: The file handle returned from an open.
       *               on_off: A 1 for on, 0 for off
       *
       * Return Value:False if operantion could not be provided,
       *               otherwise true.
       */

int
serial_send_break(int handle, int time );

      /*
       * Procedure:serial_send_break
       *
       * Function:Function to send a break signal
       *
       * Inputs:  handle: The file handle returned from an open.
       *               time  : The time in ms of the break condition
       *
       * Return Value:False if operantion could not be provided,
       *               otherwise true.
       */

/******************************* Misc. ******************************/
```

```
int
chkESC(void);

      /*
       * Procedure:   chkESC
       *
       * Function:    check to see if escape character entered at keyboard
       *
       * Parameters:
       *              NONE
       *
       * Return Values:
       *              FALSE, TRUE - if keyboard escape
       *
       * Discussion:
       *              the keyboard is read
       *          if escape entered return TRUE
       *          else return FALSE
       *
       * Imports modified:
       *              NONE
       */

int
send18xxMsg(int fd, int msgCode, char *string);

      /*
       * Procedure:send18xxMsg
       *
       * Function:Send a string message (upto the size of
       *                MAX_MSG_SZ characters) to the 18xx.
       *
       *                NOTE: If ‘string’ is greater than MAX_MSG_SZ,
       *                it is truncated to MAX_MSG_SZ.
       *
       * Inputs:  fd    - the serial port file descriptor
       *                msgCode - the message code number, as defined
       *                          in pt2.h
       *                string- the string to be sent
       *
       * Outputs: Returns the value returned by write_port(),
       *                which is one of: ERROR, TIMEOUTERR, NOERROR.
       *
       * Imports modified:none
       */

char *
getDfpConfigFile(char *buf);
```

```
      /*
       * Procedure:getDfpConfigFile
       *
       * Function:Creates the path to the dfp configuration file.
       *                This path is located in the current board
       *                directory.
       *
       * Inputs:  buf   - an empty string buffer of at least
       *                          _MAX_DIR + 1 characters.
       *
       * Outputs: buf   - contains the path to the dfp config
       *                          file
       *
       * Returned Value:address of buf, if no errors
       *                NULL, if errors
       *
       * Imports modified:none
       *
       * WARNING: This routine returns the correct path only
       *                for B.0 or latter DFP software.
       */

char *
getBoardDir(char *buf);

      /*
       * Procedure:getBoardDir
       *
       * Function:Creates the path to the current board directory.
       *
       * Inputs:  buf   - a string buffer of at least
       *                          _MAX_DIR + 1 characters.
       *
       * Outputs: buf   - contains the path of the current
       *                          board dir.
       *
       * Returned Value:address of buf, if no errors
       *                NULL, if errors
       *
       * Imports modified:none
       */

void
initDrivePath( struct dirPath *dpath );

      /*
       * Procedure:initDrivePath
       *
       * Function:Initializes the dirPath structure
       *
       * Inputs:  dpath - a pointer to a dirPath struct
       *
       * Outputs: dpath - initialize to a invalid path
       *
       * Returned Value:none
       */

int
saveDrivePath( struct dirPath *dpath );
```

```
      /*
       * Procedure:savDrivePath
       *
       * Function:Save the Current Working Directory (CWD)
       *
       * Inputs:  dpath - a pointer to a dirPath struct
       *
       * Outputs: dpath - initializes dpath to the CWD
       *
       * Returned Value:ERROR- if getcwd() failed
       *                NOERROR
       */

int
restoreDrivePath( struct dirPath *dpath );

      /*
       * Procedure:restoreDrivePath
       *
       * Function:Restores the saved path in ‘dpath’
       *
       * Inputs:  dpath - a pointer to a dirPath struct
       *
       * Outputs: none
       *
       * Returned Value:ERROR- if chdir() or _chdrive() failed
       *                NOERROR
       */
```

# ISO9141 Option 8

The ISO9141 option to the DFP responds to a customer request to use bi-directional ISO9141 logic levels in communication between the DUT and the DFP. The option kit contains a circuit board and connecting cable. The circuit board takes up a slot in the DFP with the connecting cable (a short length of 34-conductor ribbon) providing the electrical connections between the CCC and the ISO9141 option circuit board. The addition of the ISO9141 option to a CCC replaces the RS232 facilities normally present on the CCC. The addition of the ISO9141 to one CCC does not disable or change behavior of any other features on the CCC, and does not affect other CCDs in the DFP. The CCC and DR2p were originally designed with options of this nature in mind, so no modifications to the CCC or DR2p are needed. The design emphasizes the following three themes:

**1 Utility**
Two channels of ISO9141 communication are provided by one option kit. This will permit the tester to communicate with two DUT's at the same time, or to operate two sub-panels of a multipanel DUT at once.

**2 Simplicity**
Programmer selects ISO9141 as he/she would have selected RS232.

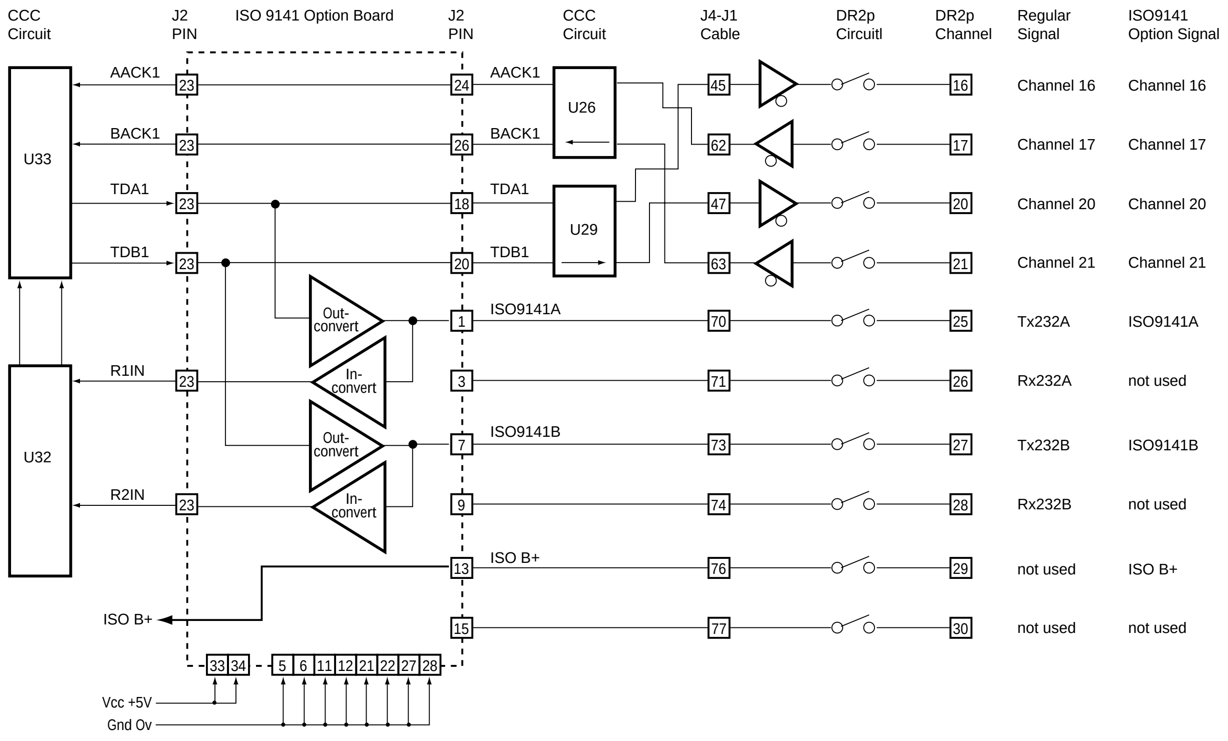
**3 Minimum impact**
RS232 is the only feature sacrificed on the CCC/DR2p pair. It can be easily restored by removing the ISO9141 option and replacing the slip-on jumpers that normally occupy positions on J2 of the CCC. Each ISO9141 option occupies one card slot in the DFP motherboard.

This document explains how the ISO9141 option works, how to program it, how to install it and how to remove it.

# Theory of Operation

The option contains two out-converters, two in-converters and some jumper connections.

Figure 8.1 Block Diagram of ISO 9141 Option to Digital Function Processor



The out-converters are potentially active all the time. They convert high CMOS logic level signals (5 volts) to high (510 ohm pulled up to B+) ISO9141 signals. The signals being converted originate in a dual UART on the CCC. They pass through GAL U33 on the way, emerging as signals TDA and TDB. Signal PCSL enables the UART signals to drive these serial channels. The full equations for this GAL can be found in the hardware section of the DFP User's Guide.

The B+ signal will normally be supplied from the DUT or DUT power supply through the fixture interface. The option commits one pin to this purpose, channel 29 of the associated DR2p. A diode to the DFP 5-volt supply provides enough B+ voltage to conduct a self test.

The in-converters, like the out-converters, are also active all the time. The inbound ISO9141 signals are first converted to RS232 levels on the option card. The RS232-level signals are then converted to 5-volt CMOS levels on the CCC by the MAX242 chip U32. Finally they pass through GAL U33 on their way to the

CCD’s dual UART. U33 uses the “SMODE” signal to decide which of two signal sources to connect to the UART. If SMODE is low, U33 selects the 5-volt logic levels AACK and BACK, which come directly from logic level pins in Group C of the DR2p. If SMODE is high, U33 selects the outputs of U32, and therefore selects the ISO9141 signals as input. For full logic equations in GAL U33, see the DFP/PROMPTest II User’s Guide.

## Programming

To enable serial transmission, program the PCSL bit to 1 via the miscellaneous register. This is necessary for logic level transmission via Group C pins on the DR2p as well as for ISO9141 transmission via Group D pins on the DR2p. In addition, for ttl, program the SHDN* bit to 0, and for ISO9141 program the SHDN* bit to 1 (disable/enable U32).

To receive ISO9141 signals through the Group D pins on the DR2p, program SMODE to 1 via the miscellaneous register. To receive logic level input signals through the Group C pins on the DR2p, program SMODE to 0. Again, for ISO9141 the SHDN* bit must be programed to 1.

### Example -- TTL level - CHA

```
outp(ca[card]|0x6006,0x21);  // misc reg:B0=1=serial mode (U33 CCC)
                             //          B3=0=disable RS232 driver (U32-CCC)
                             //          B4=0=Recieve-TTL mode (U33-CCC)
                             //          B5=1=DFP enable (to the DR2P via CCC)

outp(ca[card]|PC_CNTL,0x01); // Port C cntl-TXDA output, RXDA input
outp(ca[card]|PC_DATA,0x00); // Port C data-enable serial data mode (U73-DR2P)

outpw(ca[card]|GA_INST,D_REED|nodA[position]|SET);     // TxDA  (DR2P)
outpw(ca[card]|GA_INST,D_REED|nodA[position+1]|SET);   // RxDA  (DR2P)
```

### Example -- ISO level - CHA

```
outp(ca[card]|0x6006,0x39);       // misc reg:  B0=1=serial mode (U33-CCC)
                                  // B3=1=enable RS232 driver (U32-CCC)
                                  // B4=1=Recieve-RS232 mode (U33-CCC)
                                  // B5=1= DFP enable (to the DR2P via CCC)

outpw(ca[card]|PAB_CNTL,0xDD00);  // channel 25 aux relay (D_REED) closed for
                                  // bi-directional CHA - TXDA/RXDA (DR2P),
                                  // and channel 29 aux relay (D_REED) closed
                                  // to provide B+ from the DUT   (DR2P)
```

## System Capacity

The DFP has five card slots in all, and is intended to support up to four CCCs. Therefore, A system with four CCCs supports one ISO9141 option card (slot limit); A system with three CCCs supports two ISO9141 option cards (slot limit); A system with two CCCs supports two ISO9141 option cards (each option card needs a CCC); A system with one CCC supports one ISO9141 option card (cache option card needs a CCC).

## Installation

Install both the CCC card and ISO9141 option board into free slots in the DFP chassis. To install the ribbon-cable jumper from the ISO9141 board to the CCC card, first remove the slip-on jumpers from header J2 on the CCC, then install the ribbon-cable jumper from J2 on the CCC to J2 on the option card. Connect the cable from the CCC to DR2p and test.

## Removal

Remove the ribbon jumper from the CCC. Install slip-on jumpers on J2 of the CCC to connect pins 1 to 2, 3 to 4, 17 to 18, 19 to 20, 23 to 24, 25 to 26. Reinstall the CCC in the motherboard.

# Maintenance 9

The preliminary information included in the paper version of this manual be useful in performing product maintenance.

If you have further questions relating to the maintenance of the Digital Function Processor, please feel free to call 1-800-457-8326 (1-800-HLP-TEAM).

# Index